Differential Greedy for the 0–1 Equicut Problem

R. Battiti and A. Bertossi

ABSTRACT. Given the success of recent greedy schemes with tie-breaking rules proposed by the authors, this paper extends the investigation by considering different greedy approaches for the graph bi-partitioning problem. In particular, a new method is proposed (DIFF-GREEDY) that ameliorates in a significant way the performance of the previously proposed MIN-MAX-GREEDY, while requiring less computation. The computational complexity is demonstrated to be $O(|E|)$, E being the edge set of the graph. Experimental results are presented for a large sample of two classes of randomly generated graphs, one with a connection probability independent of the vertices, the second one with a two-dimensional geometric structure. Remarkably enough, the cut sizes obtained through independent repetitions are close to, and in some cases better than the cut sizes obtained by more sophisticated techniques based on standard greedy construction followed by local search with prohibition-based diversification (Tabu Search), for comparable running times.

Preprint UTM-515, Dip. di Matematica, Univ. di Trento, Italy
To appear in: *Proceedings of the DIMACS Workshop on Network Design: Connectivity and Facilites Location*, Princeton Univ., 1997.
Edited by: D.Z. Du and P.M. Pardalos, American Mathematical Society, 1997

1. Introduction

The 0-1 equicut 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 two disjoint subsets such that the *cut size*, i.e., the number of edges whose endpoints are in different subsets, is minimized and the difference of cardinalities between the largest and the smallest subset is at most one. Without loss of generality one can assume that the number of vertices is even. The problem is also known as *balanced* bi-partitioning, or bi-section problem.

Because of the NP-hardness of the bi-partitioning problem [**8**], heuristics have a crucial role for the solution of large-size partitioning problems in acceptable computing times. The problem has been studied extensively in the past [**2, 3, 5**]. In particular, [**9**] presents a detailed empirical study of Simulated Annealing (SA), following the application proposed in [**11, 12**]. Simple Genetic Algorithms (GA) are proposed in [**14, 15**], while a state-of-the-art GA with preliminary preprocessing is

1991 *Mathematics Subject Classification.* ...
Key words and phrases. greedy algorithms, graph-partitioning, 0-1 equicut, computer implementation .

analyzed in [**4**]. Prohibition-based approaches include the seminal Kernighan–Lin (KL) algorithm [**10**] and the more recent Tabu Search (TS) work of [**16**] and [**7**]. Greedy and randomized procedures (GRASP) are proposed in [**13**].

A very fast and effective greedy procedure (MIN-MAX-GREEDY) was recently proposed in [**1**]. In particular, *independent* repetitions of MIN-MAX-GREEDY reach competitive results on a subset of the graphs considered in [**4**]. In the same paper it is shown that, 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 heuristic with a superior performance with respect to the GA of [**4**] is obtained.

Given the success of the MIN-MAX-GREEDY algorithm, the purpose of this paper is to extend the investigation by considering different greedy approaches for the problem. In particular, a new method is proposed (DIFF-GREEDY), that ameliorates in a significant way the performance of MIN-MAX-GREEDY, particularly when independent repetitions are considered. Remarkably enough, the cut sizes obtained through independent repetitions are close to, and in some cases better than the cut sizes obtained by more sophisticated prohibition–based techniques [**7**].

In addition, while the previous experimental work was based on a limited set of representative graphs derived from [**4**], the experiments of this paper are on a large number of graphs with specific statistical properties produced by the random instance generator used in [**7**]. This method avoids the potential pitfalls related to having a small number of graphs.

The remaining part of this paper is organized as follows. The graph generators used for the experiments are described in Sec. 2. The standard greedy algorithm and the MIN-MAX-GREEDY scheme of [**1**] are briefly summarized in Sec. 3, that also contains an analysis of the number of ties in the standard greedy algorithm (Sec. 3.1) and the results obtained on the test graphs (Sec. 3.2).

The new DIFF-GREEDY algorithm is introduced in Sec. 4, its implementation and computational complexity are analyzed in Sec. 5, and the obtained experimental data about performance and CPU time are presented in Sec. 6.

2. Graph generators

The computational tests in this paper are executed on two classes of graphs: random graphs and graphs with a two-dimensional geometric structure, with a number of vertices ranging from 124 to 1000. Selected instances with the same statistical properties have been used in [**9**]. In the present work we use a random graph generator to create one hundred instances for each dimension and expected degree. The code for the generator has been obtained from the authors of [**7**] and encapsulated into a C++ class. The properties of the different graphs 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.

All the code has been developed in a high–level object–oriented language (C++). The compiler used is the `g++` compiler from GNU, the target machine for the computational tests is a Digital AlphaServer 2100 Model 5/250 with 4 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. The times listed do not include the input/output times and the times to create the initial data structures because these are the same for all the algorithms and independent of the number of iterations.

3. Min-Max Greedy

The MIN-MAX-GREEDY has been introduced in [**1**] and is summarized here to define the notation, to make this paper self-contained and to permit a comparison with the DIFF-GREEDY algorithm described in Sec. 4.

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 **if** $(in0, in1) \in E$ **then** $f \leftarrow 1$ **else** $f \leftarrow 0$
6 $tobeadded \leftarrow V \setminus \{in0, in1\}$
7 $addset \leftarrow 1$
8 **while** $|tobeadded| > 0$ **do**
9 $addset \leftarrow (1 - addset)$
10 $otherset \leftarrow (1 - addset)$
11 $minedges \leftarrow \min_{i \in tobeadded} E(i, otherset)$
12 $candidates \leftarrow \{i \in tobeadded \ : \ E(i, otherset) = minedges\}$
13(*) $maxedges \leftarrow \max_{i \in candidates} E(i, addset)$
14(*) $candidates \leftarrow \{i \in candidates \ : \ E(i, addset) = maxedges\}$
15 $bestvertex \leftarrow$ random vertex $\in candidates$
16 $addset \leftarrow addset \cup \{bestvertex\}$
17 $f \leftarrow f + minedges$
18 $tobeadded \leftarrow tobeadded \setminus \{bestvertex\}$
19 **return** f

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

Let us introduce some notation. 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:

$$E(i, set) \equiv |\{(i, j) \in E \ : \ j \in set\}| \tag{1}$$

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 7 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. Let f be the function to be minimized: $f(X) = |\{(i,j) \in E \ : \ X(j) \neq X(i)\}|$, *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 $\Delta f = E(i, otherset)$.

The standard greedy construction scheme consists of placing two random "seed" vertices into the two sets of the partition. The subsequent additions of vertices to the two sets are done such that the minimum possible increase Δf in the *cut size* f is obtained at each step. Variations of this greedy approach are used for example in [**13, 7**]. The standard greedy is described in Fig. 1, after canceling lines 13 and 14. The two random and different seed vertices are chosen and added to the two sets (lines 1–4). The initial f value is 0 if the two seed vertices are not connected, 1 otherwise (line 5). The initial set of yet unassigned vertices (*tobeadded*) contains all vertices apart from the two seeds (line 6). Then the main loop follows (lines 9–18). 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 9–10). Then the minimum number of additional edges in the cut introduced by the new addition is determined (*minedges* in line 11). Finally, the candidate vertices are determined as those producing the given *minedges* value (line 12), lines 13–14 are skipped, and a random vertex in the *candidates* set is chosen (*bestvertex*, line 15) and added to the given set (line 16). The cut size f is updated (line 17) and the vertices to be added loose *bestvertex* (line 18).

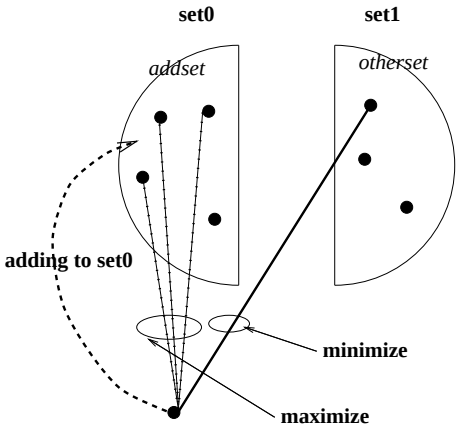

FIGURE 2. MIN-MAX-GREEDY in action: situation before adding a new vertex to $set0$.

The MIN-MAX-GREEDY algorithm adds a tie–breaking rule: among the candidates with the same *minedges* value, a subset with the largest number of edges to *addset* is extracted (lines 13–14). The criterion is illustrated in Fig. 2. A random extraction is then executed if more than one candidate survives the sieve (line 15). 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 term "Min-Max" reflects the two-step selection process, where the primary goal

is to minimize $E(i, otherset)$, the secondary one to maximize $E(i, addset)$. A fast implementation of the MIN-MAX-GREEDY algorithm is discussed in [**1**], that also presents experimental results on the suite of graphs used in [**4**].

The explanation for the poor performance of the standard greedy algorithm, demonstrated by the experimental results of Sec. 3.2, is given by the large number of ties (number of vertices producing the same Δf, or size of the *candidates* set at line 12 of Fig. 1) occurring during the construction: the algorithm is making a "blind" random choice among a sizable fraction of the nodes that are to be added. An investigation of this phenomenon is presented in the following section.

3.1. Ties in the standard greedy construction. Fig. 3 shows the average number of ties on geometric "U" graphs with expected vertex degree $d = 5$. Each curve shows the average of ten runs on different graphs with a given dimension (n = 124, 250, 500, 1000).

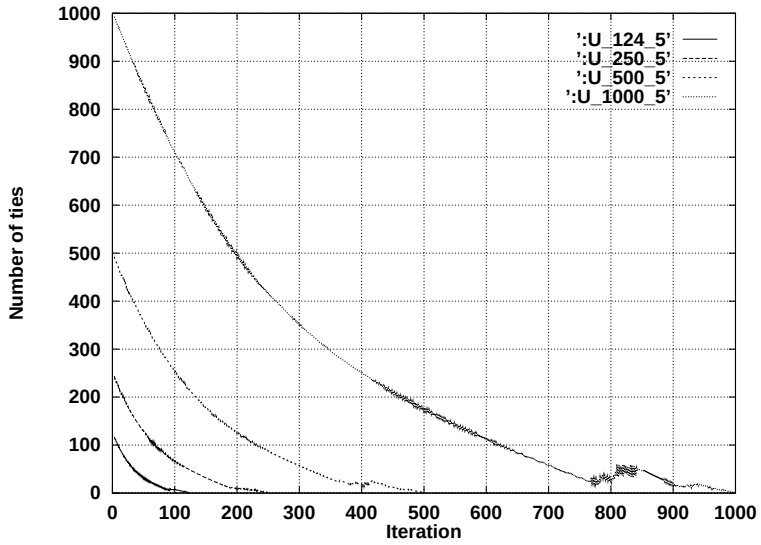

FIGURE 3. Number of ties while executing the greedy algorithm on random graphs. Un.5, for n = 124, 250, 500, 1000. Averages over 10 runs. For an easy identification, note that a line spans a number of iterations equal to the number of vertices of the graphs.

It can be noted that a very large number of ties is present, especially in the initial phase. In fact, this number corresponds to a sizable fraction of the *tobeadded* set, whose size at a given iteration t is given by $n - t$. Qualitatively similar curves are obtained for different expected vertex degrees and for the random graphs, see also the following results.

Let us now investigate the behavior in more detail. First, it is apparent that the curves for different graph dimensions look like a scaled version of the same curve with some fine details superimposed for the last iterations. To test this hypothesis it is therefore appropriate to divide both the number of ties and the iteration by the graph dimension. The results after this scaling are shown in Fig. 4, for random graphs of different dimensions and expected vertex degrees. After the scaling, the

curves become almost undistinguishable. Again, qualitatively similar results are obtained for the geometric graphs.

In Fig. 4 one observes an approximately linear decrease in the number of ties at the beginning, that becomes less rapid in the later phase, and finally some triangular patterns in the final phase. The explanation is as follows.

At iteration t, t edges have already been added. In the initial phase one can assume that most of the connections from the few nodes in $set1$ and $set0$ will be to nodes in the *tobeadded* set. In addition, because of the small number of added nodes, one can assume that most connections will be to different nodes.

Without loss of generality, let's say that the next addition is to $set0$. Because the expected degree is d, a total of approximately $(t/2)d$ nodes in the *tobeadded* set will be connected to $set1$, while $n-t-(t/2)d$ nodes are not connected and therefore can be added to $set0$ without adding any edge across the cut. In other words, the initial rate of decrease of the number of ties is given by $(1 + d/2)$. The initial behavior shown in Fig. 4 is in very close agreement with the prediction, showing that the above assumptions are approximately valid.

The rate of decrease slows down in the later phases, when there is a growing probability that some connections from $set0$ (or $set1$) are to the same vertices in the *tobeadded* set or, after one starts adding edges through the cut, to vertices in the other set.

The triangles are related to transitions in the value of *minedges*. In particular, the first "discontinuity" in the number of ties happens when *minedges* goes from 0 to 1, and even that happens sooner for the denser graphs. The approximately linear decrease after the first jump is related the decreasing occupation of the bucket containing the vertices with $minedges = 1$, until it is empty and one has a second jump related to the transition from $minedges = 1$ to $minedges = 2$, and so forth. See also Sec. 5 for the details about the buckets. Of course, some statistical noise is superimposed on the different curves, also because one alternates between additions to $set0$ and $set1$, leading to a fine saw-toothed pattern.

3.2. Experimental results: standard greedy versus MIN-MAX-GREEDY**.** Table 1 lists the average size of the cut (with its standard deviation) obtained by the MIN-MAX-GREEDY algorithm on 100 graphs of different dimensions and expected vertex degrees built by the random generator described in Sec. 2. For each graph, a total of 100 runs have been executed, by using different seeds for the random number generator used in the algorithm. The results of MIN-MAX-GREEDY are compared with those of the standard greedy scheme (middle column) and with those obtained through a random initialization. These results confirm those obtained in [**1**] on the instances used in [**4**]: a substantial reduction in the average cut size is obtained by passing from the random initialization, to the standard greedy, to the MIN-MAX-GREEDY scheme, while the CPU time of MIN-MAX-GREEDY is a small multiple of that required for a random initialization and calculation of the cut size. The obtained results will be compared with those obtained by the newly proposed DIFF-GREEDY scheme in Sec. 6.

Graph n d	**Min-Max Greedy** Ave (St.Dev.)	Greedy without Tie Breaking Ave (St.Dev.)	Random Initialization Ave (St.Dev.)
G_124_2.5	**19.6 (4.6)**	35.9 (5.6)	78.8 (10.3)
G_124_5	**73.2 (6.3)**	89.6 (7.8)	154.6 (11.4)
G_124_10	**198.9 (11.4)**	214.4 (12.1)	311.8 (17.4)
G_124_20	**470.5 (16.0)**	483.9 (16.0)	624.6 (22.6)
G_124_40	**1057.4 (21.8)**	1066.7 (21.9)	1249.5 (29.4)
G_124_80	**2303.6 (24.7)**	2308.4 (24.1)	2504.6 (34.2)
G_250_2.5	**37.6 (6.9)**	69.7 (7.8)	156.8 (12.4)
G_250_5	**145.7 (10.4)**	178.8 (11.0)	318.2 (16.8)
G_250_10	**389.4 (20.0)**	423.8 (19.9)	627.7 (24.0)
G_250_20	**929.3 (24.0)**	960.0 (24.9)	1258.5 (35.6)
G_250_40	**2074.7 (37.0)**	2097.1 (37.8)	2511.1 (45.4)
G_250_80	**4458.3 (44.5)**	4478.1 (38.8)	5023.9 (55.8)
G_500_2.5	**72.2 (9.0)**	136.4 (10.2)	312.9 (19.8)
G_500_5	**282.6 (14.6)**	353.4 (17.2)	628.7 (23.4)
G_500_10	**769.8 (22.6)**	839.4 (23.8)	1255.6 (36.9)
G_500_20	**1833.2 (38.8)**	1897.7 (39.5)	2510.8 (51.0)
G_500_40	**4079.8 (51.5)**	4142.0 (54.6)	5008.3 (72.9)
G_500_80	**8768.0 (65.3)**	8815.6 (72.8)	10009.7 (87.4)
G_1000_2.5	**143.8 (12.0)**	273.1 (14.4)	625.0 (23.4)
G_1000_5	**560.5 (21.9)**	696.2 (26.0)	1247.6 (35.1)
G_1000_10	**1522.6 (31.8)**	1668.9 (32.9)	2500.5 (49.7)
G_1000_20	**3633.9 (54.8)**	3765.8 (52.6)	4998.7 (69.3)
G_1000_40	**8108.0 (71.9)**	8231.4 (67.5)	10025.3 (91.4)
G_1000_80	**17390.9 (95.4)**	17485.2 (104.9)	19982.9 (139.2)
U_124_2.5	**1.6 (1.8)**	14.7 (4.5)	70.5 (7.5)
U_124_5	**10.4 (6.2)**	36.3 (9.0)	139.6 (12.7)
U_124_10	**35.5 (13.4)**	81.0 (21.9)	267.7 (19.6)
U_124_20	**109.5 (30.8)**	171.8 (38.2)	501.7 (29.5)
U_124_40	**298.6 (66.1)**	370.6 (83.1)	916.2 (47.0)
U_124_80	**817.3 (126.0)**	866.5 (164.4)	1616.5 (84.6)
U_250_2.5	**2.0 (1.9)**	30.4 (6.2)	148.1 (11.2)
U_250_5	**11.4 (6.3)**	70.5 (13.5)	287.7 (15.2)
U_250_10	**50.6 (17.7)**	157.3 (27.3)	566.8 (28.6)
U_250_20	**152.6 (42.4)**	322.1 (67.6)	1085.6 (45.9)
U_250_40	**451.2 (114.5)**	682.8 (152.6)	2034.2 (77.7)
U_250_80	**1258.4 (235.7)**	1550.3 (329.9)	3735.7 (137.2)
U_500_2.5	**2.0 (2.5)**	60.6 (9.7)	301.2 (16.0)
U_500_5	**12.2 (7.1)**	141.5 (17.5)	594.1 (24.0)
U_500_10	**67.1 (19.7)**	315.3 (42.0)	1159.8 (44.7)
U_500_20	**228.5 (63.0)**	666.1 (111.9)	2263.3 (54.0)
U_500_40	**690.2 (161.3)**	1305.8 (220.7)	4347.1 (106.2)
U_500_80	**1820.4 (303.3)**	2750.7 (636.2)	8179.2 (174.1)
U_1000_2.5	**1.9 (2.0)**	116.6 (12.3)	607.3 (22.6)
U_1000_5	**17.7 (8.8)**	293.0 (25.8)	1208.3 (35.0)
U_1000_10	**94.8 (30.1)**	640.2 (60.3)	2376.2 (49.2)
U_1000_20	**328.0 (75.8)**	1310.2 (139.1)	4673.4 (85.5)
U_1000_40	**965.3 (208.3)**	2614.5 (357.7)	9050.1 (138.7)
U_1000_80	**2676.7 (449.3)**	5302.4 (939.1)	17359.9 (253.7)

TABLE 1. MIN-MAX-GREEDY algorithm compared with standard greedy and random initialization. Average cut sizes (over 100 tests).

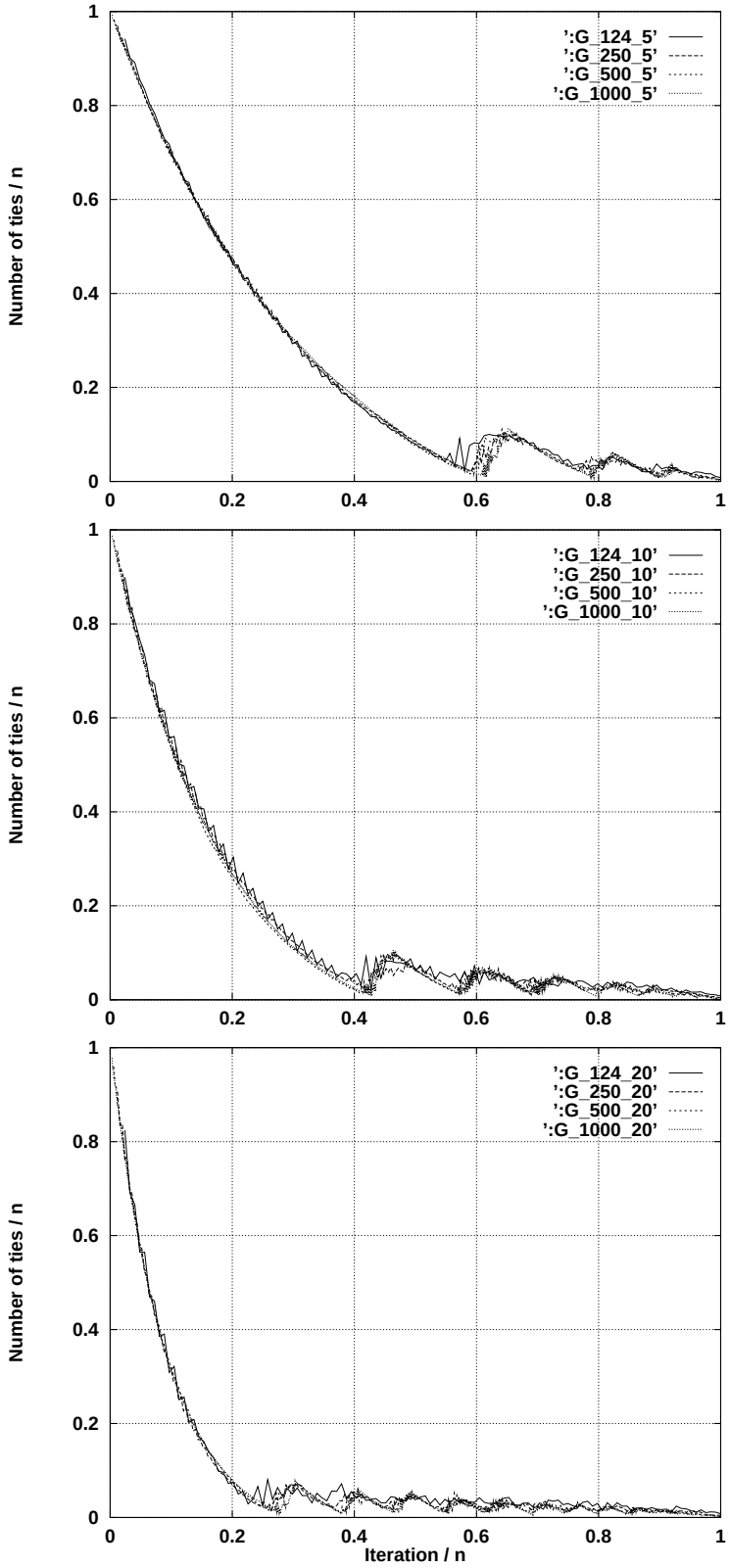

FIGURE 4. Number of ties while executing the greedy algorithm on random graphs of the kind Gn.d. Gn.5 (above) Gn.10 (middle) Gn.20 (below). The different curves are for $n = 124, 250, 500, 1000$. Averages over 10 runs.

4. Differential greedy (DIFF-GREEDY)

The MIN-MAX-GREEDY scheme aims at *minimizing* the number of new edges across the cut at each iteration, with a tie-breaking rule such that the number of new internal edges is *maximized*. The realization requires two arrays of buckets, to keep track of the number of edges between candidate vertices and nodes in the two sets $set0$ and $set1$, see [**1**].

The question we address in the present work is to assess whether a similar performance can be obtained by modifying the selection criterion in order to prefer vertices that minimize the *difference* between new edges across the cut and new internal edges. In this way, one could substitute the two arrays of buckets with a single one, in order to keep track of the differences, let us say $E(i,1) - E(i,0)$, for each candidate vertex. In this manner the same weight is given to the number of new edges across the cut ("crossing edges") and to the number of new internal edges. Of course, the criteria lead in general to different winning vertices. The experiments will clarify whether weighting in the same manner the crossing edges and the internal edges (DIFF-GREEDY) is preferable to weighting more the crossing edges (MIN-MAX-GREEDY), in fact so much more that the number of new internal edges acts only to break ties.

```
DIFF-GREEDY
1     in0 ← random vertex ∈ {1, ..., n}
2     in1 ← random vertex ∈ {1, ..., n} \ {in0}
3     set0 ← {in0}
4     set1 ← {in1}
5     if (in0, in1) ∈ E then f ← 1 else f ← 0
6     tobeadded ← V \ {in0, in1}
7     addset ← 1
8     while |tobeadded| > 0 do
9         addset ← (1 − addset)
10        otherset ← (1 − addset)
11        minedges ← min_{i∈tobeadded} E(i, otherset) − E(i, addset)
12        candidates ← {i ∈ tobeadded: E(i, otherset) − E(i, addset) = minedges}
13        bestvertex ← random vertex ∈ candidates
14        addset ← addset ∪ {bestvertex}
15        f ← f + minedges
16        tobeadded ← tobeadded \ {bestvertex}
17    return f
```

FIGURE 5. The DIFF-GREEDY algorithm.

The DIFF-GREEDY algorithm is illustrated in Fig. 5, where the new selection criterion is active in lines 11–12.

5. Implementation of DIFF-GREEDY and computational complexity

In order to implement the DIFF-GREEDY algorithm illustrated in Fig. 5 one needs to keep the values of the differences $E(i, otherset) - E(i, addset)$ up to date during the execution of the main loop (lines 9–16 in Fig. 5).

In detail, let us define as DIFF(i) the quantity $E(i,1) - E(i,0)$. When one is adding a vertex to $set0$, one needs to minimize DIFF(i) over the nodes i that have to be added. After the winning node is selected and added, its neighbors obtain an additional connection to $set0$, and therefore the corresponding values of DIFF(j) must be decreased by one.

Vice versa, when one is adding a vertex to $set1$, one needs to maximize DIFF(i) over the nodes i and, after the winning node is selected and added, the values of DIFF(j) for its neighbors must be increased by one.

A trivial realization is to use two priority queues, one for storing the "priorities" $E(i,1) - E(i,0)$, the other one for storing $E(i,0) - E(i,1)$. In practice, given the specific properties of the 0-1 equicut problem, one can use a single priority queue, such that both the minimum and the maximum of DIFF(i) are kept. Because of its double purpose the queue is termed "max-min priority queue."

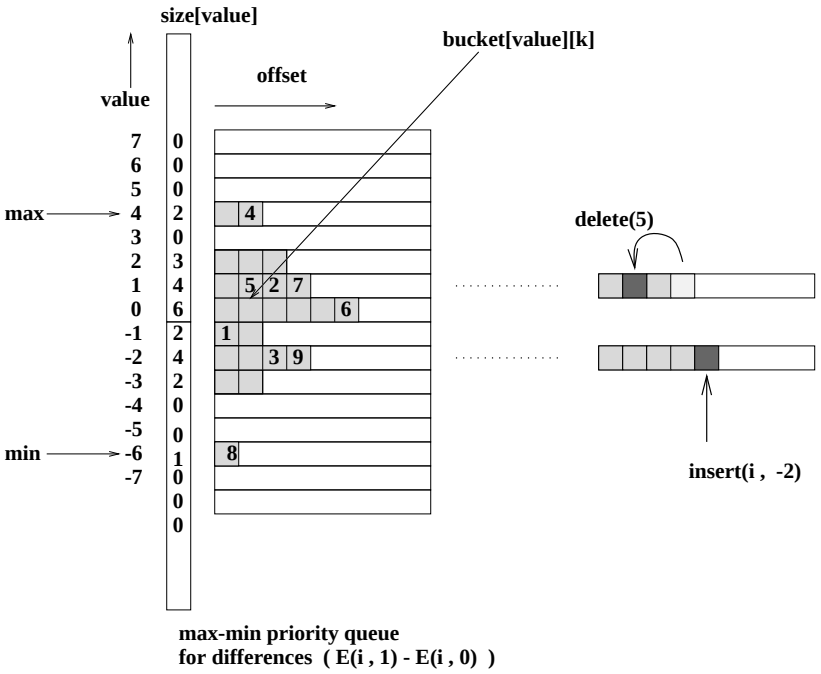

FIGURE 6. Max-min priority buckets. The operations for a generic deletion and insertion are shown in the right part.

The data structure (see Fig. 6) can be realized through a single array of buckets, indexed by the possible DIFF(i) values, clearly in the range $[-n, n]$, n being the number of nodes in the graph. Furthermore, if the maximum vertex degree deg_{max} is known, only the range $[-deg_{max}, deg_{max}]$ is actually needed. Each bucket is realized through an array of size n. The vertices i with a given $value$ of DIFF(i) are stored sequentially in $bucket[value][k]$, starting at the position $k = 0$. The size of the bucket for a given value is recorded in $size[value]$, and the offset each vertex i in the bucket is stored in the array $offset[i]$. Finally, the difference values DIFF(i) are kept in the $diffvalue$ array, and the max and min values of DIFF(i) reached by the vertices contained in the queue are kept up to date.

For example, in Fig. 6, vertex number 7 has DIFF$(7) = 1$, it is therefore stored in $bucket[1]$ together with other three vertices with the same DIFF value ($size[1] = 4$). It appears in the fourth slot and therefore one has $offset[7] = 3$.

The "max-min priority queue" permits to realize the operations needed by our algorithm in a very efficient way. In detail, in addition to the trivial operations to read the *max* and *min* values, one needs the following operations:

- INSERT(i, val): vertex i is inserted at the bucket for value $val =$ DIFF(i) in the first free array position: $bucket[val][\ size[val]\ ]$. Therefore $offset[i]$ becomes $size[val]$, then $size[val]$ is incremented by one, see the example in the right part of Fig. 6. The quantity val is saved in $diffvalue[i]$.
- DELETE(i): the vertex in the last bucket position is copied into the position containing i, then $size[$DIFF$(i)]$ is decremented.
- UPDATE$(i, delta)$: the function is called when the DIFF(i) increases ($delta = 1$) or decreases ($delta = -1$) after a neighbor of node i has been added to a set. The function is immediately realized by first deleting i from the appropriate bucket and then inserting it into the bucket corresponding to a value incremented by delta.

If the updating of the *max* and *min* values is not considered, it is trivial to check that the worst-case computational complexity of the above operations is $O(1)$. More details about the realization, that also consider the way in which the *max* and *min* values are kept up to date, are described in Sec. 5.1.

```
DIFF-GREEDY
1     ▷ add two seeds, see Fig. 5, lines 1–5
2     ▷ create random permutation of 1, .., n in permute
3     for i ← 1 to i do
4         ⌈ j ← permute[i]
5         ⌊ pqueue.INSERT(j, graph.EDGE(j, in1) − graph.EDGE(j, in0))
6     addset ← 1
7     while pqueue.SIZE > 0 do
8         ⌈ addset ← (1 − addset)
9         | otherset ← (1 − addset)
10        | if addset = 0 then opt ← pqueue.MIN else opt ← pqueue.MAX
11        | bestvertex ← pqueue.FIRST(opt)
12        | pqueue.DELETE(bestvertex)
13        | addset ← addset ∪ {bestvertex}
14        | forall j ∈ NEIGHBORS(bestvertex) do
15        |     ⌈ if X(j) = −1 then
16        |     |     ⌈ if addset = 0 then delta ← −1 else delta ← 1
17        |     |     ⌊ pqueue.UPDATE(j, delta)
18        |     ⌊ else if X(j) = otherset then f ← f + 1
19    return f
```

FIGURE 7. Implementation of the DIFF-GREEDY algorithm through the "min-max priority queue."

Let us now see how the above described data structure is used during the DIFF-GREEDY algorithm, see Fig. 7. A C++ like notation is used in the algorithm description: *pqueue* is the instance of the "max-min priority queue," and the

operations are called by appending their name separated by a single dot, like in $pqueue$.INSERT(j, val).

The two random seeds $in0$ and $in1$ are added as already explained in Fig. 5, and a random permutation of the vertices is generated at the beginning. This single randomization is useful to obtain different greedy constructions when the algorithm is repeated without incurring the computational cost to collect all candidates and to generate random numbers in the middle of the main cycle, as it was done in Fig. 5. Then the vertices are inserted into the $pqueue$ structure in the order given by the obtained permutation, and with the appropriate DIFF(j) values. The graph is described in the $graph$ data structure and $graph$.EDGE(a, b) is a member function returning 1 if edge $(a, b) \in E$, 0 otherwise, in $O(1)$ time.

The additions alternate between the two sets as described previously, until all vertices have been added. The function $pqueue$.SIZE returns the number of nodes remaining in the queue. For a single addition to $set0$ or $set1$, one picks the first vertex listed in the lowest or higher bucket, respectively (lines 10–11, Fig. 7), deletes it from the queue and adds it to the appropriate set (lines 12–13). The all neighbors j of the $bestvertex$ are considered. If j has not been included, its DIFF(j) value must be updated (lines 15–17). If j has already been added to the $otherset$, a new edge crosses the cut and the function must be updated (line 18). Finally, the cut size is returned (line 19).

Let us now consider the computational complexity of the DIFF-GREEDY algorithm. The generation of the random permutation and the insertion of the vertices cost $O(n)$, and the main loop is repeated $O(n)$ times. If one neglects the need to update the max and min values in the structure when $pqueue$.DELETE$(bestvertex)$ is called, the loop for all neighbors of the just selected vertex (lines 14–18) is the only operation that is not $O(1)$, in fact it is $O(deg_G(bestvertex))$, where $deg_G(bestvertex)$ is the vertex degree.

Now, because the range of possible DIFF values is in $[-n, n]$ and because the changes of the min and max values can be only of $+1$ or -1 during UPDATE operations, one can demonstrate that the total number of steps needed to follow the min and max values during the entire greedy construction is $O(n)$, see Sec. 5.1 for details.

After collecting the above results, and under the (standard) assumption that $|E| \geq n$, the total computational complexity results to be $O(|E|)$.

5.1. Updating max and min in the priority queue. The only non-trivial analysis to obtain the computational complexity of the DIFF-GREEDY algorithm is related to the operations needed to update the max and min values during the greedy construction. Now, updating max and min requires, in the worst case, $O(1)$ for INSERT, $O(n)$ for DELETE, and $O(1)$ during UPDATE, provided that the increments are $+1$ or -1 as it is the case for 0-1 equicut. Fortunately, it is possible to show that all the operations required during the entire greedy construction cost at most $O(n)$. In other words, updating the max and $main$ values has an *amortized cost* per iteration of $O(1)$.

Let us now analyze DELETE and UPDATE operations in detail. The dangerous event that may cause a large computational cost is when, during a call to DELETE, the bucket corresponding to a given max or min value becomes empty. In this case the corresponding value can be changed by an $O(n)$ quantity, see also Fig. 8.

```
DELETE(i)
1     value ← DIFF(i)
2     ▷ delete node i from its bucket and update size[value]
3     ▷ assume queue is not empty after deletion
4     if size[value] = 0 then
5        if value = max then
6           max ← max − 1
7           while size[value] = 0 do max ← max − 1
8        if value = min then
9           min ← min + 1
10          while size[value] = 0 do min ← min + 1
```

FIGURE 8. Updating the max and min values in the priority queue in a DELETE operation.

But let us now consider the entire sequence of updates required by the complete greedy construction. For simplicity, let us consider only the evolution of the max value (the "symmetric" demonstration can be presented for the min value) and let us assume a unit cost for each basic tracking step involving a check of the size of a bucket and a decrease (see line 7 in Fig. 8). When $set1$ and $set0$ are empty, max is equal to zero. When vertices are added to $set0$ or $set1$ at a given iteration t, the max value can either be changed by +1 or -1, if the bucket corresponding to the max value does not become empty after the addition, or suffer a drastic reduction in the opposite case. In the worst case, max may reach the lowest possible value at iteration t, i.e., $-\lceil t/2 \rceil$, trivially lower-bounded by $-n$. But, in order to fall for many steps, the max value must first "gain height" through small steps of size +1. An $O(n)$ bound on the total number of operations during the entire construction can be obtained by using *amortized analysis* techniques [**6**]. In particular, by using the "accounting" method, one may start with n credits, and "prepay" one unit of credit when max increases. The starting credit and the accumulated prepaid amount can be used when max decreases, giving a total cost of $O(n)$. A less pictorial demonstration can be given by adapting the "potential" technique: to a configuration of the max-min priority queue at the completion of iteration t, one associates a potential $\Phi(PQ_t) = n + max$. Then one defines as *amortized cost* $\hat{c}_t$ of the t-th operation the quantity:

$$\hat{c}_t \;=\; c_t \;+\; \Phi(PQ_t) \;-\; \Phi(PQ_{t-1})$$

and easily derives the bound:

$$\sum_{t=1}^{n} c_t = \sum_{t=1}^{n} \hat{c}_t \;+\; \Phi(PQ_0) \;-\; \Phi(PQ_n)$$

One can easily check that the absolute value of the difference between the initial and final potential is bounded by $O(n)$, while each amortized cost is $O(1)$, therefore the total cost is $O(n)$.

6. Experimental results of DIFF-GREEDY

The average cut sizes (with standard deviation) obtained by the DIFF-GREEDY scheme on the test graphs are collected in Table 2.

Graph n d	**Diff Greedy** Ave (St.Dev.)	Diff Greedy, 100 rep. Ave (St.Dev.)	Diff Greedy, 1000 rep. Ave (St.Dev.)
G_124_2.5	17.8 (4.4)	13.8 (3.6)	13.3 (3.4)
G_124_5	69.2 (7.0)	61.4 (6.1)	60.2 (6.1)
G_124_10	190.1 (11.6)	179.0 (10.4)	177.3 (10.3)
G_124_20	458.2 (15.3)	443.2 (14.5)	440.3 (14.2)
G_124_40	1040.6 (21.2)	1021.1 (20.5)	1017.8 (20.2)
G_124_80	2285.5 (22.4)	2268.9 (22.5)	2265.4 (22.4)
G_250_2.5	35.5 (6.8)	27.7 (5.2)	26.1 (5.1)
G_250_5	136.4 (8.2)	124.0 (8.5)	121.5 (8.0)
G_250_10	372.4 (16.5)	356.2 (16.3)	351.8 (16.4)
G_250_20	899.7 (23.0)	874.7 (21.5)	869.3 (21.6)
G_250_40	2028.1 (33.6)	1995.7 (31.6)	1988.1 (31.6)
G_250_80	4402.9 (40.9)	4364.0 (38.7)	4354.3 (38.7)
G_500_2.5	65.4 (8.7)	55.7 (7.6)	53.6 (7.4)
G_500_5	265.4 (14.3)	247.4 (11.3)	242.8 (11.4)
G_500_10	735.2 (23.1)	710.2 (20.0)	702.9 (18.9)
G_500_20	1776.0 (35.3)	1738.0 (32.5)	1729.4 (32.0)
G_500_40	3991.9 (47.2)	3941.0 (47.4)	3927.4 (48.0)
G_500_80	8644.2 (65.0)	8574.4 (64.9)	8554.8 (64.3)
G_1000_2.5	129.5 (11.4)	115.0 (9.2)	110.8 (8.8)
G_1000_5	522.9 (21.8)	496.5 (18.2)	489.3 (18.3)
G_1000_10	1454.6 (29.9)	1420.1 (25.6)	1408.3 (25.6)
G_1000_20	3511.6 (49.7)	3460.2 (44.9)	3445.6 (43.1)
G_1000_40	7914.1 (59.1)	7845.5 (59.2)	7822.7 (57.5)
G_1000_80	17099.1 (95.2)	17007.0 (88.8)	16978.0 (85.7)
U_124_2.5	1.7 (2.0)	0.0 (0.1)	0.0 (0.0)
U_124_5	8.1 (5.3)	2.3 (1.4)	2.3 (1.4)
U_124_10	32.6 (13.3)	19.5 (5.6)	19.3 (5.5)
U_124_20	108.7 (34.7)	77.2 (19.8)	77.1 (19.7)
U_124_40	288.5 (65.8)	249.7 (46.5)	249.7 (46.5)
U_124_80	803.4 (120.2)	761.0 (114.3)	760.9 (114.3)
U_250_2.5	1.5 (1.7)	0.0 (0.1)	0.0 (0.1)
U_250_5	9.8 (6.5)	2.0 (1.4)	1.8 (1.3)
U_250_10	46.2 (16.8)	24.3 (5.7)	23.9 (5.5)
U_250_20	146.1 (42.6)	103.6 (21.3)	103.0 (20.8)
U_250_40	422.0 (88.8)	353.3 (49.4)	353.2 (49.3)
U_250_80	1206.8 (204.7)	1108.6 (178.2)	1108.5 (178.1)
U_500_2.5	1.5 (1.5)	0.0 (0.0)	0.0 (0.0)
U_500_5	10.9 (6.3)	2.5 (1.6)	2.2 (1.4)
U_500_10	62.3 (19.8)	33.4 (6.1)	32.0 (5.5)
U_500_20	215.0 (59.5)	146.4 (24.2)	144.3 (23.5)
U_500_40	652.3 (144.7)	504.6 (69.6)	502.3 (68.2)
U_500_80	1785.5 (303.0)	1541.6 (206.0)	1541.2 (205.7)
U_1000_2.5	1.5 (2.2)	0.0 (0.0)	0.0 (0.0)
U_1000_5	12.1 (6.3)	2.8 (1.9)	2.3 (1.5)
U_1000_10	79.5 (25.8)	45.1 (8.3)	42.4 (7.3)
U_1000_20	302.5 (69.1)	205.1 (28.3)	198.3 (24.8)
U_1000_40	915.6 (209.7)	689.8 (85.2)	681.2 (82.5)
U_1000_80	2552.5 (407.4)	2187.5 (210.2)	2182.9 (206.6)

TABLE 2. DIFFGREEDY algorithm, for different number of repetitions. Average cut sizes (over 100 tests).

In detail, for each graph in the sample of 100 graphs with the specified dimension and expected vertex degree, a total of 100 runs are executed with different

random generator seeds. In the different tests, each run is composed of a single DIFF-GREEDY construction (first column of results in Table 2), or of the repetition of 100 independent constructions (second column) or of the repetition of 1000 constructions (third column). In the last two cases one considers in each run the *best* value obtained in all repeated constructions, and this value is then averaged over the 100 runs.

After comparing the results of Table 1 and Table 2 one derives the following conclusions. First, the average cut sizes obtained by DIFF-GREEDY are significantly better than those obtained by MIN-MAX-GREEDY, while requiring less computational effort (see below). For example, on the G_124_2.5 graphs, the cut size decreases from 19.6 to 17.8, i.e., by about 10% , and decreases of some percent points are obtained for most graphs. Furthermore, additional significant decreases are obtained by independent repetitions of DIFF-GREEDY. For example, an additional decrease of about 22 % (from a cut size of 17.8 to 13.8) is obtained with 100 repetitions on the G_124_2.5 graphs. *Vice versa*, a limited performance improvement, typically less than 1%, is obtained after passing from 100 to 1000 repetitions. In other words, 100 repetitions are sufficient to reach a performance plateau such that only minor additional improvements can be expected even after a ten-fold increase of effort.

Fig. 9 reports the average running times in *milliseconds* as a function of the average number of edges in the different classes of graphs. The four different lines in each plot correspond to graphs of different dimensions n, while the points along a given line show the running times for different expected degrees d. The linear behavior predicted by the complexity analysis is clearly visible for the considered numbers of edges and the fitted curve gives the following results: cpu time (milliseconds) $\approx$ 0.0008 $|E|$. The fine structure that is visible for the smaller graphs is probably related to use of the cache: small graphs can be places in the cache and memory access is faster. Apart from the linear increase, it is to be noted that the actual running times are very small, as it is expected from the simplicity of the algorithm and of the support data structure. In fact, more than one million edges are dealt with in one second.

When the largest G_1000_80 graphs are considered, the average CPU times are of 25 millisec for a random initialization and calculation of the cut size, of 34 millisec for DIFF-GREEDY, of 41 millisec for MIN-MAX-GREEDY. A similar relative ratio holds for all other graphs.

Given the promising performance obtained by DIFF-GREEDY, it is of interest to compare the results with those obtained by more sophisticated heuristics. In particular, the comparison is executed with the EnTaS ("Enhanced Tabu Search") algorithm of [**7**], an algorithm that can be considered the state of the art for the 0-1 equicut problem. The EnTaS algorithm incorporates repeated standard greedy constructions, local search, diversification based on prohibitions (Tabu Search) with a dynamic updating of the basic parameter (the tabu list), diversification by evaluated solutions, and periodic restarting.

The EnTaS algorithm has been run by the authors of [**7**] on the same graphs that we use in our paper, for a total of 5 seconds on a Pentium PC at 100 MHz. After scaling the time to take into account the different machine speed, as measured by the SPECint92 data, one obtains an equivalent running time of approximately 2.2 sec on our machine.

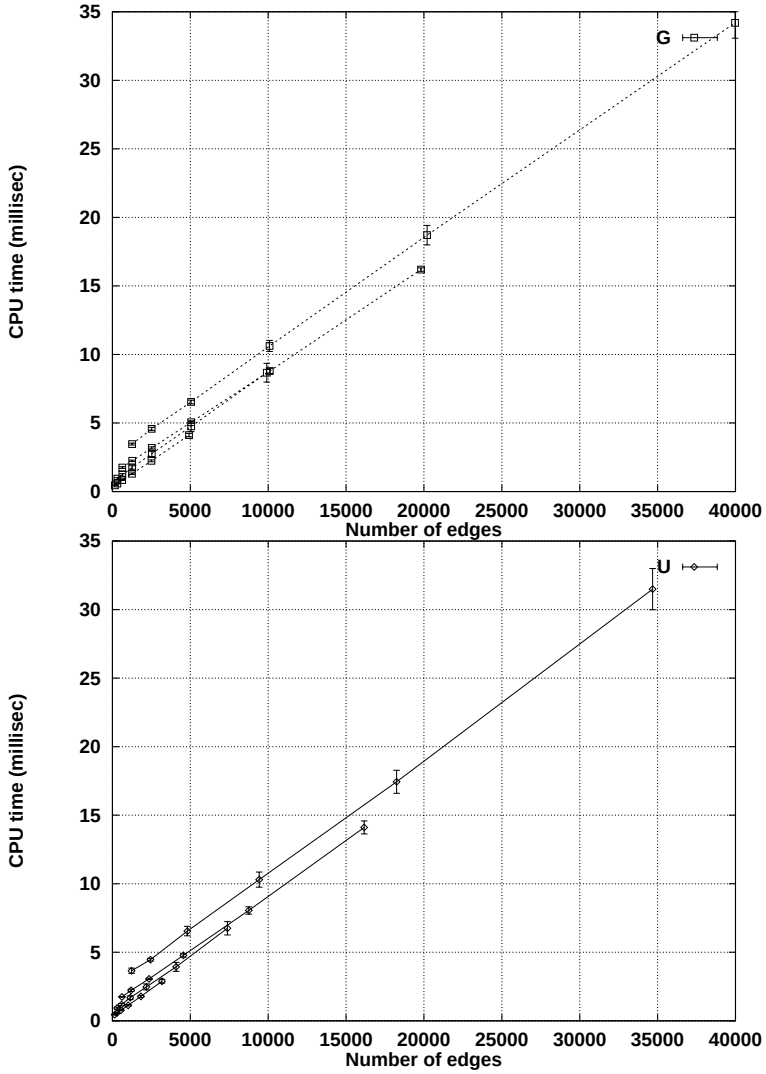

FIGURE 9. CPU times for a single run of DIFF-GREEDY. G graphs of different dimensions (above), U graphs of different dimensions (below). Averages over 1000 runs, with standard deviation bars.

Table 3 reports the results of EnTaS in the first columns, and the relative difference between DIFF-GREEDY and EnTaS in the second (for 100 repetitions) and third column (for 1000 repetitions). In detail, these columns list the quantity $100 \times (\hat{f}_{\text{DIFF-GREEDY}} - \hat{f}_{EnTaS})/\hat{f}_{EnTaS}$, $\hat{f}$ being the average cut size. The CPU times for 100 repetitions range between less than 0.3 seconds for the smallest graphs, to about 3.2 seconds for the larger ones, see also Fig. 9. Clearly, times are multiplied by ten for 1000 repetitions. One can therefore compare EnTaS results with the third column for the smallest and lowest density graphs, with the second column for the larger and denser graphs. Of course, a more detailed comparison should be based on allowing CPU times proportional to the number of edges also

for EnTaS. Nonetheless, the results in Table 3 are useful for a first comparison. It can be observed that the performance of DIFF-GREEDY tends to be within a few percent points for the random graphs, while it tends to be superior, in some cases by up to 50 - 60 % , for most geometric graphs.

The results is quite unespected, especially if one considers the simplicity of the DIFF-GREEDY scheme, with no tunable parameters, versus the sophistication of EnTaS, where eight parameters have been selected by tuning the algorithm on the given graphs.

7. Conclusions

This work extended the design and analysis of greedy algorithms for the 0-1 equicut problem. In particular, a study of the number of ties as a function of the iteration shows that the standard greedy algorithm is making a blind choice among a sizable fraction of the nodes to be added, especially for the lower-density graphs, therefore identifying a crucial reason for its poor performance. In addition, the qualitative form of the graphs has been explained.

A new algorithm is then proposed (DIFF-GREEDY), where the selection criterion during the construction aims at maximizing the *difference* between new internal edges and new edges across the cut after the addition. The focus of attention is shifted from the edges across the cut to all edges, because of the simple fact that the more edges become internal, the less edges will cross the cut in later steps of the construction.

The new algorithm is experimentally compared with a random extraction, the standard greedy scheme, the MIN-MAX-GREEDY algorithm previously proposed by the authors and a state-of-the art algorithm (EnTaS) based on local search with additional diversification mechanisms. DIFF-GREEDY clearly dominates all previous greedy algorithms (it needs less computation and provides better average cut sizes) and is comparable with the more complex EnTaS algorithm in spite of its simplicity (it is superior on the geometric graphs and inferior on the random graphs, although close results are obtained).

For a limited time, the C++ code corresponding to the algorithms described in this paper will be available from the authors for research purposes.

Acknowledgments

We would like to thank M. Dell' Amico and F. Maffioli for sending us their random instance generator and the solutions obtained by EnTaS, and D. S. Johnson for a useful discussion and for suggesting "differential" greedy approaches. This research was partially supported by Progetto Speciale "Laboratorio di algoritmica sperimentale," Università di Trento.

Graph n d	**EnTaS**	Relative difference (percent)	
		Diff Greedy, 100 rep.	Diff Greedy, 1000 rep.
	Ave (St. Dev.)	Average % difference	Average % difference
G_124_2.5	13.1 (3.3)	5.1	0.8
G_124_5	59.7 (6.1)	2.8	0.8
G_124_10	176.3 (10.2)	1.5	0.5
G_124_20	439.1 (14.1)	0.9	0.2
G_124_40	1016.3 (19.9)	0.4	0.1
G_124_80	2263.8 (22.5)	0.2	0.06
G_250_2.5	26.1 (4.9)	6.0	**-0.3**
G_250_5	117.6 (7.8)	5.4	3.1
G_250_10	345.5 (15.9)	3.1	1.8
G_250_20	860.0 (21.1)	1.7	1.0
G_250_40	1975.9 (31.6)	1.0	0.6
G_250_80	4337.6 (38.1)	0.6	0.3
G_500_2.5	54.9 (7.2)	1.4	**-3.4**
G_500_5	237.5 (11.6)	4.1	1.9
G_500_10	685.0 (20.1)	3.6	2.5
G_500_20	1699.4 (31.6)	2.2	1.6
G_500_40	3882.7 (47.0)	1.5	1.1
G_500_80	8495.2 (63.2)	0.9	0.7
G_1000_2.5	116.6 (9.4)	**-1.3**	**-4.7**
G_1000_5	494.2 (17.9)	0.4	**-1.0**
G_1000_10	1386.5 (25.8)	2.4	1.5
G_1000_20	3394.6 (45.2)	1.9	1.5
G_1000_40	7724.9 (58.5)	1.5	1.3
G_1000_80	16837.2 (84.7)	1.0	0.8
U_124_5	2.7 (1.9)	**-14.4**	**-15.8**
U_124_10	20.5 (7.5)	**-4.9**	**-5.6**
U_124_20	78.6 (16.2)	**-1.7**	**-1.8**
U_124_40	254.3 (50.4)	**-1.8**	**-1.8**
U_124_80	751.3 (110.2)	1.2	1.2
U_250_5	3.1 (1.9)	**-36.9**	**-41.3**
U_250_10	26.4 (7.3)	**-7.9**	**-9.2**
U_250_20	109.6 (22.6)	**-5.4**	**-6.0**
U_250_40	363.4 (61.2)	**-2.7**	**-2.8**
U_250_80	1081.3 (123.0)	2.5	2.5
U_500_5	4.8 (2.9)	**-48.0**	**-53.0**
U_500_10	39.7 (10.9)	**-15.6**	**-19.3**
U_500_20	143.8 (22.5)	1.8	0.3
U_500_40	474.9 (69.5)	6.2	5.7
U_500_80	1516.2 (189.0)	1.6	1.6
U_1000_5	6.9 (3.9)	**-58.6**	**-65.7**
U_1000_10	52.8 (11.7)	**-14.4**	**-19.7**
U_1000_20	195.0 (30.0)	5.1	1.7
U_1000_40	696.7 (83.3)	**-0.9**	**-2.2**
U_1000_80	2202.5 (216.7)	**-0.6**	**-0.8**

TABLE 3. Comparison between DIFF-GREEDY and EnTaS of Dell'Amico and Maffioli [7]. Average cut sizes of EnTaS and percent differences.

References

[1] R. Battiti and A. Bertossi, *Greedy and prohibition-based heuristics for graph-partitioning*, Tech. Report UTM-512, Dip. di Matematica, Univ. di Trento, Italy, Feb 1997, submitted.

[2] R. B. Boppana, *Eigenvalues and graph bisection: an average-case analysis*, Proc. 28th Symp. Foundations of Computer Science, 1987, pp. 280–285.

[3] T. N. Bui, S. Chaudhuri, F. T. Leighton, and M. Sipser, *Graph bisection algorithms with good average case behavior*, Combinatorica **7** (1987), no. 2, 171–191.

[4] T. N. Bui and B. R. Moon, *Genetic algorithm and graph partitioning*, IEEE Transactions on Computers **45** (1996), no. 7, 841–855.

[5] T. N. Bui and A. Peck, *Partitioning planar graphs*, SIAM J. Computing **21** (1992), no. 2, 203–215.

[6] T. H. Cormen, C. E. Leiserson, and R. Rivest, *Introduction to algorithms*, McGraw-Hill, 1994.

[7] 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, Kluwers Academic Publishers, 1996, pp. 361–377.

[8] M. Garey and D. S. Johnson, *Computers and intractability: A guide to the theory of np-completeness*, Freeman, San Francisco, 1979.

[9] D.S. Johnson, C. R. Aragon, L. A. McGeoch, and C. Schevon, *Optimization by simulated annealing: An experimental evaluation; part I, graph partitioning*, Operations Research **37** (1989), 865–892.

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

[11] S. Kirkpatrick, *Optimization by simulated annealing: Quantitative studies*, J. Statis. Phys. **34** (1984), 975–986.

[12] S. Kirkpatrick, C. D. Gelatt Jr., and M. P. Vecchi, *Optimization by simulated annealing*, Science **220** (1983), no. 4598, 671–680.

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

[14] H. Pirkul and E. Rolland, *New heuristic solution procedures for the uniform graph partitioning problem: Extensions and evaluation*, Computers and Ops. Res. **21** (1994), 895.

[15] 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.), Pergamon Press, Oxford, 1992.

[16] E. Rolland, H. Pirkul, and F. Glover, *A tabu search for graph partitioning*, Annals of Operations Research, Metaheuristics in Combinatorial Optimization **63** (1996).

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

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

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

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