A PROBE based heuristic for Graph Partitioning

Pierre Chardaire, Musbah Barake, and Geoff P. McKeown

Abstract— A new heuristic algorithm, PROBE_BA, based on the recently introduced metaheuristic paradigm PROBE (Population Reinforced Optimization Based Exploration) is proposed for solving the Graph Partitioning Problem. The "exploration" part of PROBE_BA is implemented using the Differential-Greedy algorithm of Battiti and Bertossi and a modification of the Kernighan and Lin algorithm at the heart of Bui and Moon's Genetic Algorithm, BFS_GBA. Experiments are used to investigate properties of PROBE and show that PROBE_BA compares favourably with other solution methods based on Genetic Algorithms, Randomized Reactive Tabu Search, or more specialized multilevel partitioning techniques. In addition, PROBE_BA finds new best cut values for 10 of the 34 instances in Walshaw's Graph Partitioning Archive.

Index Terms— Evolutionary computing, heuristic methods, graph algorithms, graph bisection, graph partitioning.

I. Introduction

LET $G = (V, E)$ be an undirected graph, where $V = \{v_1, v_2, ..., v_n\}$ is a set of n vertices, E is a set of edges connecting the vertices, and $C = (c_{ij})$ is the adjacency matrix of G, i.e., $c_{ij} = 1$ if there is an edge between i and j, otherwise $c_{ij} = 0$. The cardinality of E is denoted by e. The set of neighbours of a vertex v_i is denoted by $\Gamma(v_i)$, and is formally defined as $\Gamma(v_i) = \{v_j \mid \{v_i, v_j\} \in E\}$. The Graph Bisection Problem (GBP), also known as the Graph Bipartitioning Problem, consists of partitioning V into two disjoint subsets A and B with cardinalities differing by at most one unit while minimizing the total number of the edges connecting the vertices in the two subsets. The two subsets (A, B) form a *bisection* of G and the set of edges connecting them is called a *cut*. The cut value is calculated by

$$T(A, B) = \sum c_{ab}, \;\; a \in A \text{ and } b \in B.$$

We will assume that the number of vertices in the graph is even, as we can always add a vertex connected to no other vertices without changing the problem. In this case the subsets A and B have the same cardinality.

Problems of graph partitioning arise in various areas of computer science, including sparse matrix factorization [1], VLSI design [2], [3], parallel computing [4]–[8], and data mining [9]. In particular the bisection problem is used in VLSI circuit placement to model the placement of "standard cells" to minimize the "routing area" used to connect the cells [10] and in physics to find the ground state magnetization of spin glasses [11].

M. Barake is with the School of Business, Center for Advanced Mathematical Sciences, American University of Beirut, P.O. Box 11-0236, Bliss Street, Beirut, Lebanon.E-mail: `mbarake@gmail.com`

P. Chardaire and G. P. McKeown are with the School of Computing Sciences, University of East Anglia, Norwich, NR4 7TJ, UK. E-mail: `{pc, gmp}@cmp.uea.ac.uk`

The GBP has been extensively studied in the past [12], [13]. The problem is NP-hard [14]. Therefore, the only practical methods that have been developed for the solution of instances of non-trivial sizes are of heuristical nature. Some of these methods use general metaheuristic paradigms for combinatorial optimization such as Simulated Annealing [15], Tabu Search [16]–[18], Genetic Algorithms [19], [20], Greedy Randomized Adaptive Search (GRASP) [21]. Other heuristics are specifically designed for the particular problem being solved. The most popular of these heuristics is the Kernighan-Lin algorithm (KL) [12]. This is a group migration algorithm which starts with a bisection and improves it by repeatedly selecting an equal-sized vertex subset in each side and swapping the two sets. Such heuristics are often the basis for metaheuristic solution specializations. Recently multilevel algorithms have been developed to solve problems of very large size. These multilevel algorithms approximate the original graph 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 sequence of graphs and refined. Multilevel techniques have been proposed in [22]–[26]. Walshaw [27] makes a case for the use of multilevel refinement as a metaheuristic for the solution of combinatorial problems.

Multilevel techniques are usually much faster than metaheuristic-based algorithms but do not always compete in terms of solution quality. Some of these techniques were developed for the fast preprocessing of data in parallel calculation. The objective in some practical applications is not necessarily to find optimal solutions but to find reasonably good solutions within short computing times. The recent algorithm TPART developed by Saab [26] seems to be able to find good solutions (within a few percent of the best known solution) rapidly as well as high quality solutions (best known or very close to best known) if enough time is allocated to the method (more iterations.)

II. Motivation

Our motivation in this research was to test a new metaheuristic technique we had recently proposed. This metaheuristic christened PROBE (for Population Reinforced Optimization Based Exploration) is population based but is simpler than Genetic Algorithms as it does not include concepts of selection, mutation, and replacement. The basic idea of PROBE is to use a population to determine subspaces that are explored to find optimized solutions. These optimized solutions form a new population and the process is repeated.

Our interest in the GBP stems from the work of Bui and Moon [19] and the work of Battiti and Bertossi [18], [28] on this problem. In particular, Bui and Moon designed a GA to

which our method could be compared, and Battiti and Bertossi designed a greedy algorithm useful to the implementation of a PROBE based solution method.

The Breadth First Search Genetic Bisection Algorithm (BFS_GBA) of Bui and Moon [19] solves the GBP by using a hybridized, steady-state, genetic algorithm with multi-point crossover. Solutions are coded as bit strings with a one-to-one correspondence between bits and vertices. The value of a bit in the string indicates whether a vertex is in one side of the bisection or in the other, thus a string that has as many zeros as ones lead to a feasible solution. The fitness of each chromosome varies linearly with the cut value of the solution. At each iteration a biased selection towards the fitter chromosomes is performed to pick two parents from the population. Two offspring solutions are generated by first applying crossover and mutation. This generally leads to infeasible solutions that are repaired by using a simple scheme of flipping bits. The better of the two solutions is then passed to a linear time implementation of the Kernighan-Lin algorithm. The final solution obtained replaces the closest parent with respect to its Hamming distance, but only if it has a better solution value. The algorithm stops when 80% of the solutions in the population have the same solution value. An important aspect of Bui and Moon's algorithm is a preprocessing phase that uses breadth first search to reorder the vertices in an attempt to ensure that clusters of tightly connected vertices are included in short schemas that have more chance to survive a crossover operation.

Battiti and Bertossi proposed a number of Tabu Search algorithms for the GBP [18]. Their first algorithm, Fixed Tabu Search (FIXED-TS), starts from an initial bisection and improves upon it by a scheme of local search and prohibition. Details about FIXED-TS and discussion of the relationship between prohibition and KL can be found in [17]. At each iteration, a vertex is selected from one side of the bisection, moved to the other side, and is prohibited to move back again for a certain number of iterations (prohibition parameter). The input bisection to FIXED-TS is obtained by running Battiti and Bertossi's Min_Max greedy construction algorithm which builds a bisection by alternatingly adding one vertex to each partition in a greedy manner. A number of experiments were carried out and suggested that the prohibition parameter had a big effect on the quality of the solution obtained but unfortunately there appeared to be no trivial way of obtaining the best parameter value for each instance of the graphs. To solve this problem Battiti and Bertossi proposed a randomized version where the problem of parameter tuning is solved by repetitive call to the FIXED-TS algorithm with the prohibition parameter generated at random at each call. Finally, in their Reactive Randomize Tabu Search (RRTS) algorithm they used a combination of randomization and reactive approaches. Basically the method exploits information from previous runs to self-tune its parameter of prohibition. (see [18] for details.)

In Section III we present the PROBE metaheuristic. In Section IV we describe our specialization of PROBE to the solution of the GBP. Finally in Section V we test our method experimentally on benchmark problem instances and compare it with Bui and Moon's BFS_GBA, Battiti and Bertossi's

RRTS, and Saab's TPART.

III. PROBE: POPULATION REINFORCED OPTIMIZATION BASED EXPLORATION

PROBE is a population based metaheuristic that has recently been proposed by Barake, Chardaire and McKeown [29]. PROBE directs optimization algorithms, general or specific, towards good regions of the search space using some ideas from genetic algorithms (GAs) [30]. An important property of the (standard n-point) crossover operator used in GAs is that if two parents share a common bit value their offspring inherits it. PROBE uses this characteristic to generate *feasible* offspring from *feasible* parents. PROBE maintains, at each generation g, a population of feasible solutions $S_i^g, i = 0, 1, \ldots, P - 1$. The next generation of solutions is obtained as follows. For each $i = 0, \ldots, P - 1$ the solution, S_i^{g+1}, is computed from the pair (S_i^g, S_{i+1}^g) (where the subscripts are taken modulo P) by the sequence of steps in Figure 1. In this figure the term *vector of components* represents any

---

Input: A population of solutions represented as a vector of components $S_i^g, i = 0, \ldots, P - 1$.

Output: A population of solutions represented as a vector of components $S_i^{g+1}, i = 0, \ldots, P - 1$.

$S_P^g \longleftarrow S_0^g$.
for i **from** 0 **to** $P - 1$ **do**

1) Fix the components that have the same value in S_i^g and S_{i+1}^g.

2) Find an instantiation of the remaining components using an appropriate search algorithm.

3) Use the instantiation obtained as a starting point for a local optimizer.

    The solution obtained is S_i^{g+1}.

end do
return $(S_i^{g+1}, i = 0, \ldots, P - 1)$.

Fig. 1.   Generation of solutions in PROBE

appropriate coding of the solution, such as bit string or table of integers. It is clear that step 1 guarantees that the subspace searched in step 2 contains a feasible solution if both parent solutions are feasible. The PROBE basic scheme can be refined to allow control of the size of the space searched in step 1 (see [29] for details.) However these refinements are not used in our implementation for the GBP. Details of our implementation for the GBP are provided in Section IV.

A. A Theoretical Property of PROBE

An interesting feature of the PROBE meta-heuristic is that *when the selected subspace search algorithm is an exact algorithm*, the above version of PROBE guarantees that the average fitness of the solutions in the pool increases until all solutions have the same fitness.

Proposition 1: Let F_i^g be the fitness (objective value to be maximized) of solution of index i mod P in generation g and

let

$$A^g = \frac{1}{P} \sum_{i=0}^{P-1} F_i^g$$

be the average fitness of the pool of P solutions in generation g. If *the subspace search algorithm used by PROBE is an exact algorithm* then $A^{g+1} > A^g$, unless all solutions at generation g have the same fitness.

Proof: Assume with no loss of generality that $F_0^g \neq F_1^g$. If $F_0^g < F_1^g$ then $F_0^{g+1} \geq \max(F_0^g, F_1^g) > F_0^g$. Moreover, $F_i^{g+1} \geq \max(F_i^g, F_{i+1}^g) \geq F_i^g$ for $i = 1 \ldots P-1$. Hence, $A^{g+1} > A^g$.

If $F_0^g > F_1^g$ then $F_0^{g+1} \geq \max(F_0^g, F_1^g) > F_1^g$. Moreover, $F_i^{g+1} \geq \max(F_i^g, F_{i+1}^g) \geq F_{i+1}^g$ for $i = 1 \ldots P-1$. Hence, $A^{g+1} > A^g$. ∎

Corollary 1: Under the hypothesis of proposition 1 PROBE converges to a population of solutions which all have the same fitness.

Proposition 1 gives some theoretical foundation to the method even if practical implementations of PROBE do not generally meet the hypothesis of proposition 1. Our PROBE implementation for the GBP does not meet the hypothesis of proposition 1 as it does not use an exact algorithm for subspace search. However, we shall see in Section V-B that its runs converge for the instances tested.

IV. A PROBE BASED SOLUTION METHOD FOR THE GBP

In this section we detail our implementation of PROBE for the GBP (PROBE_BA for PROBE Bisection Algorithm.)

GeneratePopulation (S^g)

Input: A population of solutions represented as a vector, S^g, of bit strings $S_i^g, i = 0, \ldots, P-1$.

Output: A population of solutions represented as a vector, S^{g+1} of bit strings $S_i^{g+1}, i = 0, \ldots, P-1$.

$S_P^g \longleftarrow S_0^g$;
for i **from** 0 **to** $P-1$ **do**

    1) $(A_i, B_i) \leftarrow$ Decode (S_i^g);
       $(A_{i+1}, B_{i+1}) \leftarrow$ Decode (S_{i+1}^g);
       $A \leftarrow A_i \cap A_{i+1}$; $B \leftarrow B_i \cap B_{i+1}$;

    2) /* construction phase */
       $(A, B) \leftarrow$ DifferentialGreedy (A, B);

    3) /* local optimization phase */
       $(A, B) \leftarrow$ BuiAndMoonKL (A, B);
       $S_i^{g+1} \leftarrow$ Encode (A, B);

end do
return (S^{g+1});

Fig. 2.   Algorithm Generate Population: Generation of solutions in PROBE_BA

Each solution to the GBP is represented by a binary string which corresponds to a bisection of the graph. The number of bits in the solution equals n, the number of vertices in the graph. Each bit corresponds to a vertex in the graph. A bit has a value 0 if the corresponding vertex is in one side of the bisection, and has a value 1 otherwise. One of the bits, say,

the first one could be fixed to one to eliminate symmetrical solutions. Preliminary experimental results indicated however that it is slightly better not to use this scheme. The fitness is the cost of external links of the bisection and is to be minimized.

PROBE_BA starts with a population, S^0, of feasible solutions generated randomly and independently. The solutions are placed in the population in the order they are generated (i.e., solution of index 0 is generated first, solution of index $P-1$ is generated last.) PROBE_BA iterates the algorithm in Figure 2 for a maximum number of generations (user parameter MAXITER) or until no improvement on the best cut value has been found for a number of generations (user parameter STUCK.)

The function Decode (s) decodes a binary string, s, into an array of vertex labels whereas the function Encode performs the inverse operation. In the following subsections we examine the implementation of steps 2 and 3 of the algorithm.

A. Construction of a subspace solution

An interesting feature of the GBP is that any two solutions share as many bits fixed to one as they do bits fixed to 0. More formally we have the following proposition:

Proposition 2: Assume that (A, B) is a bisection of a graph, G, where vertices in A are labelled with 0 and vertices in B are labelled with 1. In the same way, assume that (C, D) is a bisection of G where vertices in C are labelled with 0 and vertices in D are labelled with 1. Then the number of vertices with the same label 0 in the two bisections is equal to the number of vertices with the same label 1 in the two bisections.

Proof:

$$|A \cap C| + |A \cap D| = |A| = n/2 \Rightarrow |A \cap C| = n/2 - |A \cap D|$$

and

$$|B \cap D| + |A \cap D| = |D| = n/2 \Rightarrow |B \cap D| = n/2 - |A \cap D|$$

The result follows. ∎

We determine a solution in the subspace defined by the two parent solutions by using a greedy algorithm (if the two parent solutions are identical there is only one such solution.) As shown by proposition 2, a partially constructed solution to the bisection problem with as many vertices on each side of the partial bisection can be found by fixing the vertices corresponding to the bits shared by the two parent solutions. The corresponding partial bisection is used as input to the *Differential-Greedy algorithm* of Battiti and Bertossi [28] to produce a bisection of the full graph.

The basic idea of the Differential-Greedy algorithm is to construct a bisection (A, B) by adding vertices alternately to the two sets in a greedy fashion. At each stage the vertex selected to enter a set, say the set A, is the vertex for which the number of neighbour vertices in A (internal degree) minus the number of neighbour vertices in B (external degree) is maximum. If several vertices are candidates to enter a set one of these vertices is selected at random. An outline of the algorithm is presented in Figure 3. The rationale behind the vertex selection criterion is that a bisection that minimizes

the cut size also maximizes the number of internal edges. In large graphs with low density the choice criterion used is better than one solely based on the external degree of the vertices, in particular if the algorithm is initialized with a small partial bisection. Indeed, in such a case many vertices have an external degree of zero for a large number of iterations of the algorithm.

Input: A graph, $G = (V, E)$, with $|V| \mod 2 = 0$.
A partial bisection (A_0, B_0) with $|A_0| = |B_0|$

Output: A bisection (A, B) of G.

Initialize bisection with $A \leftarrow A_0$ and $B \leftarrow B_0$,;
Initialize the set of remaining vertices: $R \leftarrow V - (A \cup B)$;

while $(R \neq \emptyset)$ **do**

Select a vertex, $x \in R$, that maximizes
$\sum_{a \in A} C_{xa} - \sum_{b \in B} C_{xb}$;

$R \leftarrow R - \{x\}$; $A \leftarrow A \cup \{x\}$;

Select a vertex, $x \in R$, that minimizes
$\sum_{a \in A} C_{xa} - \sum_{b \in B} C_{xb}$;

$R \leftarrow R - \{x\}$; $B \leftarrow B \cup \{x\}$;

end while

return (A, B);

Fig. 3.   The Differential-Greedy algorithm.

Note that the Differential-Greedy algorithm as presented in [28] starts with one vertex in each side of the bisection (these two vertices are selected at random.) In our solution method we use the Differential-Greedy algorithm to construct solutions from partial bisections.

The Differential-Greedy algorithm is also implemented using a bucket array structure. It is not difficult to establish that when this structure is used the computational complexity of the algorithm is $O(e)$ (see [28] for example.)

B. Local Optimizer

Once a solution has been constructed an improvement phase is applied. In our solution method we use a local optimizer designed by Bui and Moon [19] for their Genetic Algorithm. This local optimizer is based on the Kernighan and Lin algorithm (KL) [12].

The main feature of KL is its capability of escaping local optima by accepting non-improving moves, only if they contribute later to a better cut value. KL takes as input an initial bisection generated at random and improves upon it by swapping equal-sized subsets to create a new bisection. The process is repeated on the new bisection until no improvement can be obtained. Given a bisection (A, B), KL orders the elements of A and B using the following method: For every pair of vertices $(a, b) \in A \times B$ let $g(a, b)$ represent the reduction in cut value obtained upon swapping the two vertices. It is easy to see that $g(A, B) = D_a + D_b - 2c_{ab}$ where D_x represents the cut value reduction obtained when x is moved to the other side of the partition. D_x is called the

difference value (or D value) of vertex x. KL selects the pair (a_1, b_1) which maximizes $g(a, b)$. Once a_1 and b_1 are selected, they are assumed to be exchanged and not considered any more for further exchange. The process is repeated to produce a sequence of pairs $(a_1, b_1), \ldots, (a_{n/2-1}, b_{n/2-1})$. Then KL selects $t \in \operatorname{argmax}_k R(k)$ where $R(k) = \sum_{i=1}^{k} g(a_i, b_i)$. If $R(t) > 0$ an improved bisection, $((A - X_t) \cup Y_t, (B - Y_t) \cup X_t)$ with $X_t = \{a_1, \ldots, a_t\}$ and $Y_t = \{b_1, \ldots, b_t\}$, is obtained.

KL has a theoretical worst complexity of $O(e.n^3)$ as the maximum number of passes that might be considered is bounded by e (since each pass reduces the cut size by at least one unit and the size of the initial bisection is at most e) and the complexity of each pass is $O(n^3)$. Nevertheless, researchers have empirically shown that the number of passes does not exceed 10 in most cases, thus the typical practical complexity of KL is $O(n^3)$ (see [19] for more details).

The basic idea of the local optimizer proposed by Bui and Moon had already been proposed by Kernighan and Lin [12]. Instead of examining all pairs of vertices eligible for exchange at each iteration of a pass of KL only a small number of pairs are examined using the following method: Two vertices with largest D values in A are selected from that set. Then two vertices are selected from B in a similar way. The best pair amongst the four resulting combinations is selected for exchange. Bui and Moon use a bucket array structure to store vertices according to their D values. This reduces the complexity to $O(e)$. In addition they perform a single pass of this algorithm and limit the number of swapped pairs to a parameter. The rational is to avoid premature convergence in their GA. They found that setting the parameter to $\lfloor n/6 \rfloor$ gave the most desirable overall performance for their algorithm on the graphs they tested. In all our experiments we use the same choice of the parameter.

V. Numerical Experiments

In all our experiments the CPU times of our algorithm is given for a DELL Optiplex GX260, with Pentium 4, 2.391 GHz processor and 670 MB of usable RAM. Our programs are written in C++ and compiled under Microsoft Visual C++ using the standard release option (i.e., with optimizations performed on a per module basis only.) We compare our results with those published by Bui and Moon [19], by Battiti and Bertossi [18], and by Saab [26]. CPU times for Bui and Moon's BFS_GBA, Battiti and Bertossi's RRTS, and Saab's TPART were obtained on a Sun Sparc IPX, a Digital AlphaServer 2100 Model 5/250, and a Compaq Alpha DS20E 67/667 Mhz respectively. Battiti and Bertossi estimated that their machine is 12.7 times faster than Bui and Moon's. Our machine's architecture is similar to that of a Dell Precision Worstation 350, 2.4 GHz P4, whose SPECint2000 base rate is 9.05 which makes it approximately 2.7 times faster than a Dell precision 420 dual 733 MHz processor with 512 Mbytes of RAM. The SPECint95 benchmark indicates that the latter is 5.9 faster than Battiti and Bertossi's machine. Hence our machine is approximately 16 times faster than Battiti and Bertossi's, and 203 times faster than Bui and Moon's. The SPECint2000 base rate for Saab's machine is 4.93 indicating

that our machine is approximately 1.8 times faster than his. These coefficients have been used to scale other authors' computing times in our tables of results.

Our main interest is in establishing whether a PROBE-based solution method can compete in terms of solution quality rather than in finely comparing computing times, as long as the computing time is reasonable and does not grow too fast with the problem instance size.

The computational tests are executed on three groups of graphs.

The first group is composed of 16 graphs proposed by Johnson et al. [15] and 24 graphs proposed by Bui and Moon [19]. It contains the following types of graphs:

- *Gn.d*: A random graph with n vertices, where an edge between any two vertices is created with probability p such that the expected degree, $p(n-1)$ of a vertex is d,
- *Un.d*: A random geometric graph with n vertices uniformly distributed in the unit square. Two vertices are connected with an edge if and only if their Euclidean distance is less than or equal to $\sqrt{d/(n\pi)}$. d 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 [31].
- *cat.n*: A caterpillar graph with n vertices. It is constructed by starting with a path (the spine.) Each vertex on the spine is then connected to six new vertices, the legs of the caterpillar. With an even number of vertices on the spine, the optimal bisection size is 1.
- *rcat.n*: A caterpillar graph with n vertices, where each vertex on the spine has $\sqrt{n}$ legs. These 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.

The second group contains graphs related to applications in parallel computing. It contains four numerical grids (`airfoil1`, `big`, `wave`, `nasa4704`), two graphs that belong to the Harwell-Boeing collection [32] (`bcspwr09`, `bcsstk13`), and two De Bruijn networks (`DEBR12`, `DEBR18`.) These graphs can be obtained from the Graph Partitioning Repository at the University of Paderborn (Internet address: `http://wwwcs.uni-paderborn.de/fachbreich/AG/monien/RESEARCH/PART/graphs.html`.) Their dimensions are shown in Table I.

Finally the last group is composed of the 34 graphs in Walshaw's Graph Partitioning Archive [33] at the University of Greenwich (Internet address: `http://staffweb.cms.gre.ac.uk/%7Ec.walshaw/partition/`.) They represent a sample of real-world applications from various origins.

The dimensions of the graphs are shown in Table II.

In Section V-A we examine the impact of the strict ordering of solutions in the populations of PROBE_BA. In Section V-B we experimentally establish the convergence of the algorithm. Section V-C shows the influence of the population size on solution quality. Section V-D examines whether it is a better

Graph	Number of vertices	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

TABLE I

PARALLEL COMPUTING BENCHMARK GRAPHS

strategy to perform a small number of runs with a large population or a large number of runs with a small population. In Section V-E we compare our results with those obtained by BFS_GBA, RRTS and TPART. Finally in Section V-F we provide results for the instances in Chris Walshaw's Graph Partitioning Archive.

A. Impact of The Population Production Strategy

In the PROBE heuristic the population at generation g is generated from the population at generation $g - 1$ by the following scheme:

$$S_i^g = F(S_i^{g-1}, S_{i+1}^{g-1}) \qquad i = 0 \ldots P - 1$$

where F denotes the function that takes as input two parent solutions and returns a child solution, and the indices of solutions are taken modulo P.

We have tried to vary the strategy that produces children from parents by randomly shuffling the population after the production of each generation. Formally, let π_g be a random permutation of $0, 1, \ldots, P - 1$ used at generation g. In the version with random shuffling we apply the scheme

$$S_{\pi_g(i)}^g = F(S_i^{g-1}, S_{i+1}^{g-1}) \qquad i = 0 \ldots P - 1$$

Table III gives the average and minimum solution values over 20 runs with a population of 200 solutions. Each run is stopped when the minimum value does not improve for 100 generations (parameter STUCK in table.) The columns labelled *Iter* give the average number of generations of a run. Best solutions are highlighted in boldface. The average quality of the solutions found using the no-shuffling strategy is better than when using the shuffling strategy except for U1000.05 and U1000.10. In addition the shuffling strategy fails to find the best known values in a number of cases.
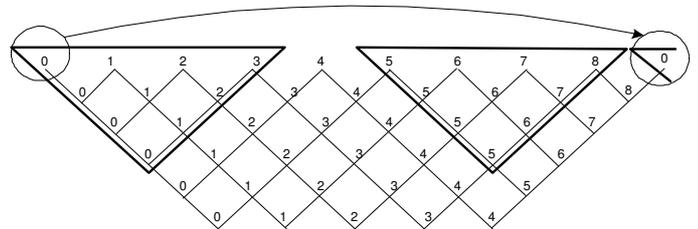

Fig. 4.   Evolution of population in PROBE_BA

One possible explanation of the success of the no-shuffling strategy is illustrated in Figure 4. Solutions that are far

Graph	Number of vertices	Number of edges	Graph	Number of vertices	Number of edges
add20	2395	7462	bcsstk30	28924	1007284
data	2851	15093	bcsstk31	35588	572914
3elt	4720	13722	fe_pwt	36519	144794
uk	4824	6837	bcsstk32	44609	985046
add32	4960	9462	fe_body	45087	163734
bcsstk33	8738	291583	t60k	60005	89440
whitaker3	9800	28989	wing	62032	121544
crack	10240	30380	brack2	62631	366559
wing_nodal	10937	75488	finan512	74752	261120
fe_4elt2	11143	32818	fe_tooth	78136	452591
vibrobox	12328	165250	fe_rotor	99617	662431
bcsstk29	13992	302748	598a	110971	741934
4elt	15606	45878	fe_ocean	143437	409593
fe_sphere	16386	49152	144	144649	1074393
cti	16840	48232	wave	156317	1059331
memplus	17758	54196	m14b	214765	1679018
cs4	22499	43858	auto	448695	3314611

TABLE II

INSTANCES IN WALSHAW'S GRAPH PARTITIONING ARCHIVE

	Without shuffling			With shuffling			Best
	Average	Min	Iter	Average	Min	Iter	known
G500.2.5	**49.1**	**49**	157.1	52.25	51	109.75	49
G500.05	**218**	**218**	117.6	219.4	218	113.4	218
G500.10	**626.7**	**626**	129.55	630.2	627	108.95	626
G500.20	**1744**	**1744**	124	1749.8	**1744**	110.6	1744
G100.2.5	**94.55**	**93**	188.45	99.45	96	115.5	93
G1000.05	**447.35**	**445**	202.25	452.8	450	117.2	445
G1000.10	**1362.65**	**1362**	174.9	1366.15	1363	117.55	1362
G1000.20	**3383.5**	**3382**	161.45	3387.4	3385	118.7	3382
U500.05	**2**	**2**	122.6	2.1	**2**	105.8	2
U500.10	**26**	**26**	104.3	**26**	**26**	103.4	26
U500.20	**178.1**	**178**	102.55	178.25	**178**	102.35	178
U500.40	**412**	**412**	102	**412**	**412**	102	412
U1000.05	1.2	**1**	131.1	**1**	**1**	106.1	1
U1000.10	39.05	**39**	111.3	**39**	**39**	106.4	39
U1000.20	**222**	**222**	104.25	**222**	**222**	103.7	222
U1000.40	**737**	**737**	102.25	**737**	**737**	102.1	737

Population = 200, STUCK = 100, 20 runs per instance

TABLE III

COMPARISON OF PERFORMANCES OF POPULATION PRODUCTION STRATEGIES

apart in the population evolve independently for a number of generations (provided the solutions in the starting population are produced independently.) More precisely, for a population of P solutions indexed from 0 to $P-1$ we define the distance between solution of index i and solution of index j as the length of the shortest path between i and j in the cycle $0, 1, 2 \ldots P-1, 0$. This distance is $\min(|i-j|, P-|i-j|)$. Two solutions at distance d evolve independently for $d-1$ generations, as illustrated by the two triangles highlighted in Figure 4. In Figure 4 solution of index 0 and solution of index 5 are at distance 4 in a population of size 9 and evolve independently for 3 generations. This indicates that a diverse population may be maintained for longer when using a no-shuffling strategy. As a consequence runs with the no-shuffling strategy may require more iterations than runs with the shuffling strategy but may also converge to better solutions. This seems to be confirmed by the results in Table III, as the number of iterations for the no-shuffling strategy is always higher than the number of iterations for the shuffling strategy, and the average and minimum cut values obtained for the harder Gn.d instances are always better. For the Un.d instances that are easy to solve there is not a huge difference between strategies as the best cut value is found early on by both methods. However for the three most difficult instances to solve, which are G500.2.5, G100.2.5 and G1000.05 (see Table V in Section V-E), the effect is more pronounced.

B. Convergence of The Algorithm

In Section III-A we showed that PROBE converges to a population of solutions with identical values when an exact method is used to generate child solutions. We shall call the value to which solution values converge *the convergence value*.

In our practical implementation children solutions are generated from parent solutions heuristically. Nonetheless convergence also happens as shown by experiments illustrated
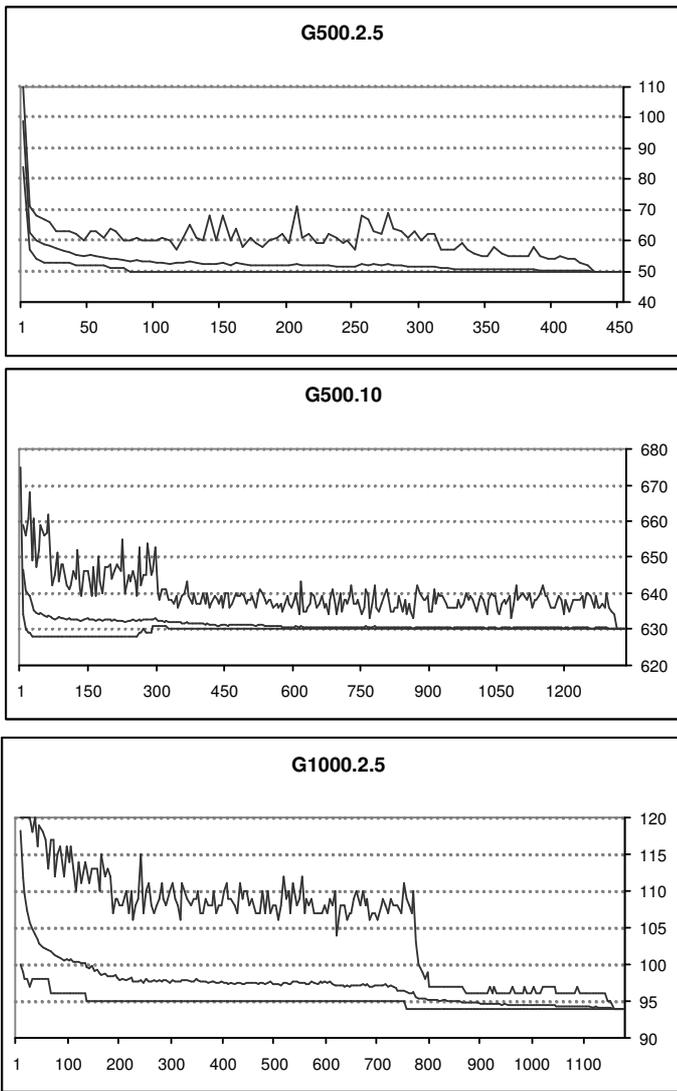
Fig. 5. Maximum, average, and minimum solution values over the population versus generation number for *Gn.d* graphs

Fig. 6. Maximum, average, and minimum solution values over the population versus generation number for *Un.d* graphs

in Figures 5 and 6 for a sample of the graphs. See the Appendix for a complete set of figures for all *Gn.d* and *Un.d* graphs. These figures correspond to experiments with a population of 50 solutions. They provide graphs of the *maximum*, *average*, and *minimum* solution values *over the population* at every five generations. Each experiments is terminated when the three values are identical for twenty five consecutive generations. The minimum cut value found in an experiment is usually found rapidly, the *average* solution value tends to decrease smoothly, while diversity in the population is maintained for a large number of iterations as shown by the more erratic behavior of the *maximum* value. The convergence to a population where all solutions have identical values is a striking feature of our experiments. Observe that sometimes the minimum value increases. There is no reason this should not happen as a good solution may be lost from one generation to the next. In general the convergence value is close to the optimum value, but it is by no means always the case. The graph for U1000.10 is a particularly interesting "bad case".
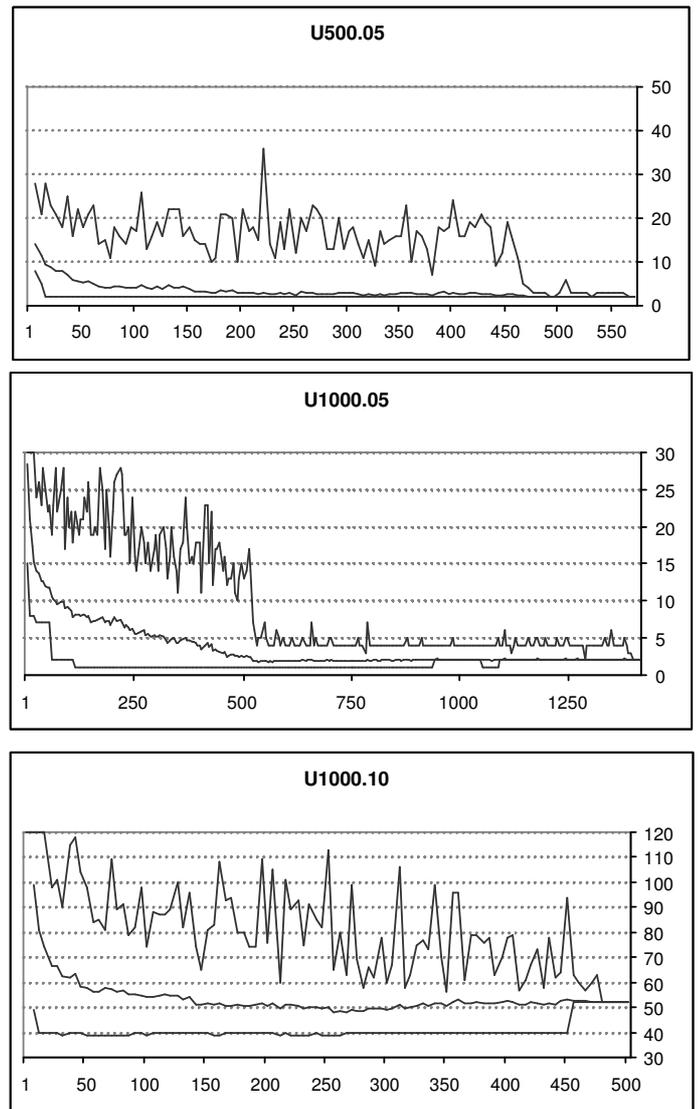
The minimum value of 39 is found at generation 31 (this is the best know solution value for this instance.) The minimum value then oscillates between 39 and 40 until generation 456 when it becomes 52 after which both *maximum*, *average*, and *minimum* values converge rapidly to 52. An identical behaviour is observed if we increase the population to 200. In this case the value 39 is obtained at generation 16. The same oscillations of the minimum value between 39 and 40 are observed until generation 456 when the minimum becomes 51 and remains so until convergence. Obviously, this does not affect the effectiveness of our method as we tend to find the minimum cut value before convergence occurs. The graph for instance G1000.2.5 is also interesting. The minimum value of 95 is found at generation 136 and remains so for 584 generations until an improved value of 94 is found. The convergence value is 94 in this experiment. In practice we do not wait for convergence to terminate a run of our algorithm. Instead we stop when no improvement of the minimum value

has occurred for a number of iterations (parameter STUCK.) We use a relatively small value of the parameter STUCK in our experiments in order to limit computing time. This can however be at the detriment of the quality of the final solution for some runs as indicated by the above observation.

C. Influence of The Population Size
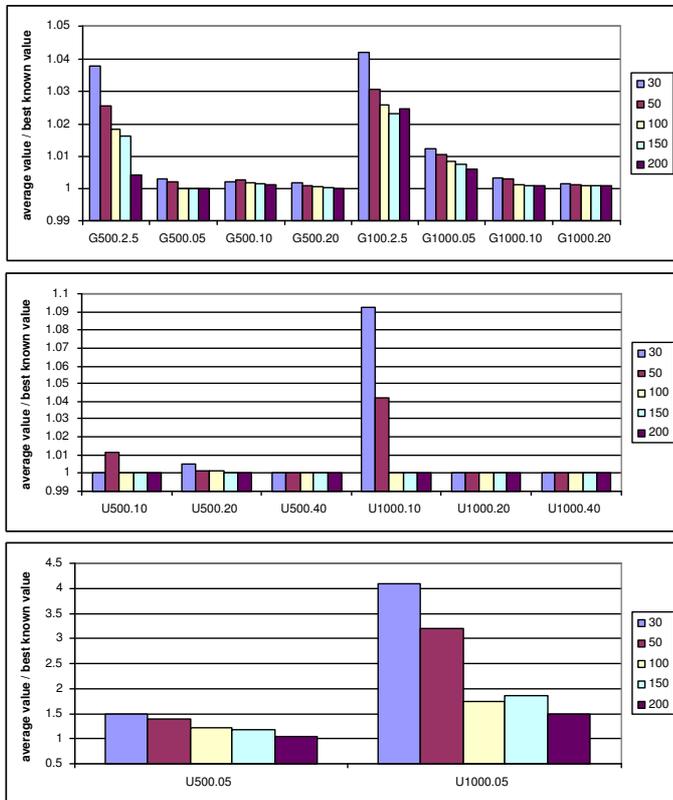
Fig. 7. Influence of population size on quality of solutions, 20 runs per size

In Figure 7 we display the quality of the average cut value found over 20 runs for various population sizes. In these experiments the value of the parameter STUCK is 50 (we stop the method after 50 generations without improvement of the minimum cut value found.) For normalization purposes the average cut value is divided by the best known cut value shown in the last column of Table III. As the population increases the average cut value decreases. To get solution values consistently close to the optimum cut value a population of at least 50 solutions should be used. Note that the optimum cut value for U1000.05 is 1 which explains the greater ratios observed. For this instance a ratio of 1.5 indicates that the optimal cut value is found in at least 50% of the runs.

D. Many Short Runs or a Small Number of Long Runs?

Should we perform a small number of runs with a large population of solutions or a large number of runs with a small population of solutions? We address this question in this section. A *multi-run* is produced by performing a number r of runs of the algorithm. The output of a multi-run is the best cut value found in the r runs of the algorithm. In Table IV we give

the minimum, maximum, average of this output and average CPU time of a multi-run over 5 multi-runs of the algorithm. The computational time of a run with a population of size kP may be assumed to be roughly k times the computational time of a run with a population of size P. This assumption is not entirely correct as the algorithm does not stop after a fixed number of generations but when the minimum value has not been improved for a number of consecutive generations. However, the assumption seems to be reasonable without any prior knowledge for an input instance. So as we divide the population by 2 we multiply r by 2. The average times of multi-runs displayed in the table confirm our assumption.

The results in Table IV indicate that to perform a small number of runs with a large population is a slightly better strategy than performing a large number of runs with a small population both in terms of minimum cut value and maximum cut value. It should also be noted that our current implementation is not particularly memory-consuming. Each individual in the population is represented in compact form as a bit string. We also keep the difference values (see Section IV-B) of the vertices in the corresponding bisection. These difference values are exploited to avoid initialization from scratch of the bucket array structure when the Differential-Greedy algorithm is applied (see Section IV-A.) So as our implementation stands the memory complexity associated with the population is $O(nP)$.

E. Performance Quality

In Table V, Table VI and Table VII we compare the performance of our algorithm with results obtained with Bui and Moon's GA [19], Battiti and Bertossi's RRTS [18] (see Sections II for a brief presentation of these algorithms) and Saab's partitioning method TPART [26]. According to Bui and Moon's results their algorithm dominates the simulated annealing algorithm presented in [15] as well as a multi-start Kernighan and Lin algorithm.

For each method we provide, whenever available, the average, standard deviation, minimum and maximum cut values over a number of runs for each instance considered.

We ran PROBE_BA with a population of 200 solutions, and a value of 100 for the parameter STUCK; and performed 40 runs with different random initializations for each graph. Results in the column "TPART(5) 100 runs" correspond to 100 runs of the method TPART with $5n$ moves in its main algorithm, and are those published in [26]. Results reported for the other techniques correspond to 1000 runs per instance. Results in columns "BFS_GBA 1000 runs" are those published in [19]. Results in columns "RRTS 1000 runs" are those published in [18].

The best results are displayed in boldface. All CPU times are average CPU times of a single run.

The minimum values returned by PROBE_BA for the *Gn.d* and *Un.d* graphs are the best known values for these instances. In fact, for G1000.2.5 we improve on the best value found by the other methods. The computing time of a run with PROBE is much larger than with the other methods. However, the method delivers quality results. All runs of PROBE_BA

	Population=200, $r=4$				Population=100, $r=8$				Population=50, $r=16$			
	Min	Max	Ave	CPU	Min	Max	Ave	CPU	Min	Max	Ave	CPU
G500.2.5	49	49	49	14.2	49	49	49	15.0	49	49	49	15.2
G500.05	218	218	218	13.6	218	218	218	15.0	218	218	218	16.2
G500.10	626	627	626.2	19.9	626	627	626.6	22.5	626	627	626.6	22.6
G500.20	1744	1744	1744	30.5	1744	1744	1744	32.3	1744	1744	1744	34.2
G100.2.5	93	94	93.8	34.4	94	95	94.2	37.3	94	94	94	38.0
G1000.05	445	446	445.2	46.7	445	448	446	43.2	445	448	446.2	46.0
G1000.10	1362	1362	1362	54.0	1362	1362	1362	60.7	1362	1362	1362	64.8
G1000.20	3382	3384	3382.8	77.6	3383	3384	3383.4	82.2	3382	3384	3382.8	87.9
U500.05	2	2	2	12.3	2	2	2	12.3	2	2	2	12.3
U500.10	26	26	26	12.1	26	26	26	12.2	26	26	26	12.7
U500.20	178	178	178	19.5	178	178	178	20.0	178	178	178	20.3
U500.40	412	412	412	25.5	412	412	412	25.6	412	412	412	25.9
U1000.05	1	1	1	25.7	1	1	1	30.2	1	1	1	30.8
U1000.10	39	39	39	28.0	39	39	39	31.8	39	39	39	33.7
U1000.20	222	222	222	45.8	222	222	222	52.6	222	222	222	46.9
U1000.40	737	737	737	57.7	737	737	737	60.8	737	737	737	61.6

min, max, average cut and cpu over 5 multi-runs per instance
r = number of runs in a multi-run; STUCK = 100

TABLE IV

COMPARISON OF RUN STRATEGIES

	BFS_GBA 1000 runs			RRTS 1000 runs ($1000n$ iterations per run)			
Graph	Ave	Min	CPU[1]	Ave	St.dev.	Min	CPU[1]
G500.2.5	53.97	**49**	0.029	52.06	0.50	51	0.125
G500.05	222.13	**218**	0.040	218.29	0.46	**218**	0.156
G500.10	631.46	**626**	0.058	**626.44**	0.59	**626**	0.225
G500.20	1752.51	**1744**	0.106	1744.36	0.66	**1744**	0.425
G1000.2.5	103.61	95	0.083	98.69	1.01	95	0.406
G1000.05	458.55	447	0.117	450.99	1.43	**445**	0.406
G1000.10	1376.37	**1362**	0.183	1364.27	1.38	**1362**	0.581
G1000.20	3401.74	**3382**	0.307	3383.92	1.00	**3382**	0.919
	TPART(5) 100 runs			PROBE_BA 40 runs[†]			
Graph	Ave	Min	CPU[1]	Ave	St.dev.	Min	Max	CPU
G500.2.5	53.21	50	0.032	**49.12**	0.33	**49**	50	3.59
G500.05	223.42	**218**	0.072	**218**	0.00	**218**	218	3.54
G500.10	631.44	**626**	0.133	626.85	0.36	**626**	627	4.90
G500.20	1751.88	**1744**	0.289	**1744.07**	0.47	**1744**	1747	7.51
G1000.2.5	99.64	94	0.089	**94.62**	0.70	**93**	96	8.86
G1000.05	459.69	448	0.200	**447.5**	1.30	**445**	450	12.0
G1000.10	1372.76	1363	0.400	**1362.4**	0.59	**1362**	1364	14.3
G1000.20	3403.28	**3382**	0.728	**3383.52**	0.82	**3382**	3385	21.2

1: Original CPU time normalized to discount the different speed of the machines
†: MAXITER = 200, STUCK = 100

TABLE V

COMPARISONS BETWEEN SOLUTION METHODS

provide solutions close to the best found, except for instance U1000.05. For this instance we found the optimal cut value 1 in 32 of the runs and values greater than 2 in only 4 runs. The worst cut values found by our method are in all cases but one better than the average cut values obtained with BFS_GBA. Our average cut value is also most of the time less than or equal to that of RRTS but RRTS gives good quality results in shorter computing time. More importantly, for the difficult instances, we obtained better results than RRTS. For instance G500.2.5 our maximum cut value is less than the minimum cut value obtained by RRTS in 1000 runs. In fact in 40 runs we got cut values of 49 (35 times) and 50 (5 times). For instance G1000.2.5, the best cut value found by BFS_GA and RRTS

in 1000 runs is 95. PROBE_BA obtained cut values of 93 (1 time), 94 (10 times), 95 (21 times), 96 (8 times). TPART finds good cut values but sometimes fails to find the minimum cut value. The average values also indicate that the method is not as consistent as PROBE_BA. However the computing times are very low. So the consistency (average cut value) can be improved by using a multi-run strategy (see [26] for details.)

For the *breg.n.b*, *grid.n.b* and *w-grid.n.b* instances our algorithm performs well in terms of quality. These instances are however also well solved by the much faster methods, BFS_GBA and TPART. For the caterpillar graphs our algorithm has difficulties in finding good solutions when the graph size increases. This may be because we use a version of KL

	BFS_GBA 1000 runs			RRTS 1000 runs ($1000n$ iterations per run)			
Graph	Ave	Min	CPU[1]	Ave	St.dev.	Min	CPU[1]
U500.05	3.65	**2**	0.037	**2**	0	**2**	0.106
U500.10	32.68	**26**	0.047	**26**	0	**26**	0.169
U500.20	179.58	**178**	0.057	**178**	0	**178**	0.331
U500.40	412.23	**412**	0.049	**412**	0	**412**	0.638
U1000.05	1.78	**1**	0.087	**1**	0	**1**	0.263
U1000.10	55.78	**39**	0.152	**39.03**	0.19	**39**	0.394
U1000.20	231.62	**222**	0.162	**222**	0	**222**	0.781
U1000.40	738.10	**737**	0.182	**737**	0	**737**	1.519

	TPART(5) 100 runs			PROBE_BA 40 runs[†]				
Graph	Ave	Min	CPU[1]	Ave	St.dev.	Min	Max	CPU
U500.05	2.34	**2**	0.024	2.05	0.22	**2**	3	3.17
U500.10	28.35	**26**	0.061	**26**	0	**26**	26	3.11
U500.20	179.05	**178**	0.161	**178**	0	**178**	178	5.10
U500.40	**412**	**412**	0.261	**412**	0	**412**	412	6.79
U1000.05	1.06	**1**	0.043	1.4	0.96	**1**	5	6.98
U1000.10	42.71	**39**	0.150	**39**	0	**39**	39	7.41
U1000.20	227.88	**222**	0.333	**222**	0	**222**	222	11.7
U1000.40	745.34	**737**	0.750	**737**	0	**737**	737	15.3

1: Original CPU time normalized to discount the different speed of the machines
†: MAXITER = 200, STUCK = 100

TABLE VI

COMPARISONS BETWEEN SOLUTION METHODS

	BFS_GBA 1000 runs			TPART(5) 100 runs			PROBE_BA 40 runs[†]		
Graph	Ave	Min	CPU[1]	Ave	Min	CPU[1]	Ave	Min	CPU
breg500.0	**0**	**0**	0.011	**0**	**0**	0.002	**0**	**0**	0.610
breg500.12	**12**	**12**	0.019	**12**	**12**	0.021	**12**	**12**	1.12
breg500.16	**16**	**16**	0.020	16.02	**16**	0.022	**16**	**16**	1.09
breg500.20	**20**	**20**	0.021	20.10	**20**	0.024	**20**	**20**	1.09
breg5000.0	**0**	**0**	0.198	**0**	**0**	0.024	**0**	**0**	9.60
breg5000.4	**4**	**4**	0.212	**4**	**4**	0.244	**4**	**4**	14.6
breg5000.8	**8**	**8**	0.221	**8**	**8**	0.244	**8**	**8**	14.9
breg5000.16	**16**	**16**	0.260	**16**	**16**	0.272	**16**	**16**	14.3
cat.352	2.25	**1**	0.011	**1**	**1**	0.007	**1**	**1**	0.85
cat.702	2.43	**1**	0.031	**1**	**1**	0.013	2.55	**1**	1.69
cat.1052	2.44	**1**	0.049	**1**	**1**	0.021	2.4	**1**	3.26
cat.5252	2.63	**1**	0.329	**1**	**1**	0.100	8.65	5	24.1
rcat.134	1.35	**1**	0.003	**1**	**1**	0.003	**1**	**1**	0.27
rcat.554	1.99	**1**	0.015	**1**	**1**	0.010	1.15	**1**	1.23
rcat.994	2.14	**1**	0.031	**1**	**1**	0.017	1.25	**1**	2.42
rcat.5114	2.36	**1**	0.259	**1**	**1**	0.089	2.57	2	19.9
gridl00.10	**10**	**10**	0.002	**10**	**10**	0.005	**10**	**10**	0.27
grid500.21	21.02	**21**	0.017	**21**	**21**	0.026	**21**	**21**	1.1
grid1000.20	**20**	**20**	0.037	**20**	**20**	0.054	**20**	**20**	2.27
grid5000.50	**50**	**50**	0.301	**50**	**50**	0.328	**50**	**50**	14.0
w-grid100.20	**20**	**20**	0.003	**20**	**20**	0.006	**20**	**20**	0.296
w-grid500.42	**42**	**42**	0.023	**42**	**42**	0.029	**42**	**42**	1.23
w-grid1000.40	**40**	**40**	0.052	**40**	**40**	0.056	**40**	**40**	2.40
w-grid5000.100	100.03	**100**	0.464	**100**	**100**	0.350	**100**	**100**	15.2

1: Original CPU time normalized to discount the different speed of the machines
†: MAXITER = 200, STUCK = 100

TABLE VII

COMPARISONS BETWEEN SOLUTION METHODS

in our improvement phase; KL is known to perform poorly for caterpillar graphs [34]. We use the same version of KL as BFS_GA but the breadth first search preprocessing of BFS_GA is the main factor in the success of the GA for these instances. Without the preprocessing the best cut values found by Bui and Moon's GA for identical experiments is 69 for cat.5252 and 11 for rcat.5114 (see [19].) TPART is the best of the methods tested for solving caterpillar graphs. The benefit of multilevel methods that may coarsen the graph by collapsing the legs of the caterpillar is clear.

	RRTS ($100n$ iterations per run)			
Graph	Ave	St.dev.	Min	CPU[1]
airfoil1	74.8	1.1	**74**	0.41
big	141.3	2.2	**139**	2.1
wave	8950.8	74.7	8835	288
bcspwr09	9.3	0.6	**9**	0.12
bcsstk13	**2355**	0	**2355**	0.73
nasa4704	1296.0	7.0	**1292**	1.2
DEBR12	558.0	1.7	556	0.91
DEBR18	24088.8	114.2	23996	3537

	TPART(5,20)			PROBE_BA, Population=50[†]				
Graph	Ave	Min	CPU[1]	Ave	St.dev.	Min	Max	CPU
airfoil1	**74**	**74**	9.3	**74**	0	**74**	74	8.6
big	**139**	**139**	40.9	157.6	17.38	**139**	187	42.8
wave	8727.2	8704	4508	8717.8	10.31	**8701**	8737	1480
bcspwr09	**9**	**9**	1.2	**9**	0	**9**	9	3.0
bcsstk13	**2355**	**2355**	34.3	**2355**	0	**2355**	2355	11.0
nasa4704	**1292**	**1292**	54.6	**1292**	0	**1292**	1292	12.7
DEBR12	**548**	**548**	13.5	**548**	0	**548**	548	9.14
DEBR18	23486	**23208**	6533	23651	246.57	23442	24226	2340

	PROBE_BA, population=200[†]				
Graph	Ave	St.dev.	Min	Max	CPU
airfoil1	**74**	0	**74**	74	30.9
big	141.2	5.9	**139**	158	164
wave	**8709.1**	4.3	8702	8715	5729
bcspwr09	**9**	0	**9**	9	11.7
bcsstk13	**2355**	0	**2355**	2355	41.3
nasa4704	**1292**	0	**1292**	1292	51.1
DEBR12	**548**	0	**548**	548	32.7
DEBR18	**23438.2**	32.8	23352	23470	9804

1: Original CPU time normalized to discount the different speed of the machines
†: MAXITER = 250, STUCK = 100

TABLE VIII

COMPARISONS ON "PARALLEL COMPUTING" GRAPHS OVER 10 RUNS FOR EACH METHOD

Table VIII compares PROBE_BA, RRTS, and Saab's multi-level algorithm TPART(5,20) (each run is the best of 20 runs of TPART(5)) over 10 runs of each method on the graphs of Table I. All CPU times are average CPU times of a single run. The results for RRTS are those published in [18]. The results for TPART are those published in [26]. The results of Battiti and Bertossi also showed that RRTS gives at least as good or better cut values than state-of-the-art partitioning software MeTis, Chaco, Scotch and Party for the instances tested excepted for DEBR12 where Party with Helpful Sets found a minimum cut value of 548 (see [18] for details.) Best results are shown in boldface in the table. PROBE_BA was run with populations of 50 and 200 solutions with, in each case, a maximum number of iterations of 250, and the parameter STUCK set to 100. PROBE_BA always find as good or better minimum cut values than RRTS. For instances wave and DEBR12 the worst result (column Max) of PROBE_BA for both population sizes is better than the best result (column Min) found by RRTS. This also applies to DEBR18 when PROBE_BA is run with a population of 200 solutions. There is not much difference in terms of quality of solutions between TPART and PROBE_BA; both methods find good solutions and produce low average cuts in similar computing time (after re-scaling). TPART finds a better cut value than PROBE_BA

with population 200 for DEBR18 but for both DEBR18 and wave the maximum cut value found by PROBE_BA is lower than the average cut value found by TPART, which indicates a greater consistency in the quality of results with PROBE_BA for these instances. PROBE_BA with both population 50 and 200 finds a better cut value than TPART for wave. Note also that in the experiments of Section V-F below we report a cut value of 8692 edges for this instance.

Saab points out that it is possible to obtain high quality solutions with his method much faster. He mentions that he obtained a cut value of 23650 edges for DEBR18 in 116.7 seconds (equivalent to 64.8 seconds on our machine) by running TPART(1,1). However he does not provide information that would indicates whether this is a typical run or the best run amongst many. The latter hypothesis seems more likely given the results of TPART(1,1) on smaller graphs.

By running PROBE_BA with a population of 50, and the parameter STUCK set to 5 (we stop as soon the minimum cut value does not steadily decrease) we obtained the following results for the more difficult instances:

	PROBE_BA 10 runs, population=50[†]				
Graph	Ave	St.dev.	Min	Max	CPU
wave	8858.8	49.8	8823	8967	256
DEBR18	24049.8	286.5	23698	24572	810

†: MAXITER = 100, STUCK = 5

		PROBE_BA, 20 runs[†]				
Graph	Best	Min	Ave.	St.dev.	Max	CPU
add20	599	**596**	598.25	0.97	600	8.8
data	190	**189**	190.8	2.63	198	16.6
3elt	**90**	**90**	90	0	90	18.2
uk	**20**	**20**	25.4	3.27	32	24.1
add32	**11**	**11**	12.9	3.02	20	23.0
bcsstk33	**10171**	**10171**	10171	0	10171	123.5
whitaker3	**127**	**127**	127.1	0.31	128	32.1
crack	**184**	**184**	184.05	0.22	185	51.1
wing_nodal	**1707**	**1707**	1708.05	1.54	1713	111.3
fe_4elt2	**130**	**130**	130	0	130	41.8
vibrobox	**10343**	**10343**	10343	0	10343	119.6
bcsstk29	**2843**	**2843**	2843	0	2843	93.7
4elt	**139**	**139**	140.9	4.39	158	89.6
fe_sphere	**386**	**386**	386	0	386	79.4
cti	**334**	**334**	334	0	334	55.5
memplus	5538	**5513**	5576.85	56.63	5689	120.0
cs4	**372**	**372**	402.6	40.67	548	199.1
		PROBE_BA, 10 runs[‡]				
Graph	Best	Min	Ave.	St.dev.	Max	CPU
bcsstk30	**6394**	**6394**	6394	0	6394	198.4
bcsstk31	2768	**2762**	2827.6	197.33	3389	201.5
fe_pwt	**340**	**340**	340.4	0.84	342	104.9
bcsstk32	**4667**	**4667**	5122.8	469.29	5811	337.4
fe_body	**262**	266	342.1	76.21	467	186.4
t60k	**80**	423	531.4	75.38	662	181.2
wing	**791**	809	998.6	107.96	1118	370.4
brack2	**731**	**731**	731	0	731	188.8
finan512	**162**	**162**	226.8	83.65	324	211.0
fe_tooth	**3850**	3897	3977.1	86.61	4102	332.8
fe_rotor	2103	**2098**	2099.1	1.28	2101	567.0
598a	2411	**2398**	2400	1.49	2402	808.4
fe_ocean	465	**464**	473.7	8.47	495	589.7
144	6491	**6490**	6497.9	6.70	6510	1871
wave	8704	**8692**	8707.1	6.43	8714	1239
m14b	**3836**	3837	3841.8	2.25	3845	2117
auto	10173	**10117**	10198.7	59.48	10304	5908

†: Population = 100, MAXITER = 200, STUCK = 100
‡: Population = 50, MAXITER = 200, STUCK = 100

TABLE IX

PERFORMANCE OF PROBE_BA ON WALSHAW'S GRAPH PARTITIONING ARCHIVE INSTANCES

This shows that our method finds good quality solutions rapidly. The relative gap between the *average* cut values shown above and the best cut values displayed in Table VIII is 1.8% (3.6%) for wave (DEBR18). The minimum cut values are also better than those found by RRTS in Table VIII.

F. Solution of the Walshaw Graph Partitioning Archive instances

Table IX gives results for the instances in Walshaw's Graph Partitioning Archive. For the smaller instances we performed 20 runs with a population of 100 solutions whereas for the larger instances we performed 10 runs with a population of 50 solutions. In both cases we limited the number of generations to 200. For each graph we give the minimum, average, standard deviation, and maximum of the best cut value found over the number of runs performed. CPU is the computing time for one run. Column "Best" is the best known solution value. Best known cut values have been obtained by various researchers using approaches ranging from KL based

partitioning techniques to semi-definite programming (see the archive for details.)

PROBE_BA does not find the best known cut value for 5 instances. It seems that our algorithm does not work well with instance t60k. t60k is a 2D Finite Element Method dual graph where the maximum vertex degree is 3 and the minimum vertex degree is 1 Other instances in the archive are also dual graphs (e.g. uk, cs4, wing.)

PROBE_BA find the best cut value previously reported for 19 instances, and an improved cut value for 10 instances. Note that the results for wave in this group of experiments are slightly better than those reported in Table VIII (the difference in CPU per run is due to a different choice of the MAXITER parameter.)

PROBE_BA's consistency varies according to the instances, as shown by the maximum cut values and the standard deviations. The population used for the larger instances was however relatively small. For fe_rotor and 598a all 10 runs provide an improvement on the previously best known

cut value.

VI. Conclusion

The main objective of our research was to test the recently proposed metaheuristic PROBE on an important combinatorial optimization problem.

PROBE_BA, our specialization of PROBE for the solution of the GBP, enabled us to establish a number of interesting properties that would be worth investigating further with other combinatorial optimization problems: The impact of the population is significant as runs with large population sizes give better solutions than runs with small population sizes. The strict ordering of solutions which maintains a degree of independence between solutions for a number of generations, is an important factor in the success of the method. Populations seem to converge to solutions of identical objective values even when there is no enforced requirement that a solution should have a better fitness than its parent solutions.

Importantly, we showed that PROBE_BA can compete with other algorithms based on Genetic Algorithms, Reactive Tabu Search, or more specialized multilevel partitioning techniques. PROBE_BA also finds improved best cut values on a number of real world instances.

PROBE_BA uses standard graph partitioning methods to explore search spaces and perform local optimization, namely the Differential-Greedy algorithm of Battiti and Bertossi and a modification of KL due to Bui and Moon. It would be interesting to investigate whether PROBE can enhance the power of other graph partitioning algorithms. Another interesting line of investigation would be to develop a PROBE-based algorithm for the solution of k-way partitioning problems. In this case several bits per vertex could be used to code the indices of the partitions to which vertices are assigned. The search of a subspace would require a partitioning algorithm that can construct as good a solution as possible from a partially initialized solution (i.e., vertices pre-assigned to partitions.)

Appendix I

The set of convergence graphs mentioned in Section V-B is provided for the *Gn.d* and *Un.d* instances.
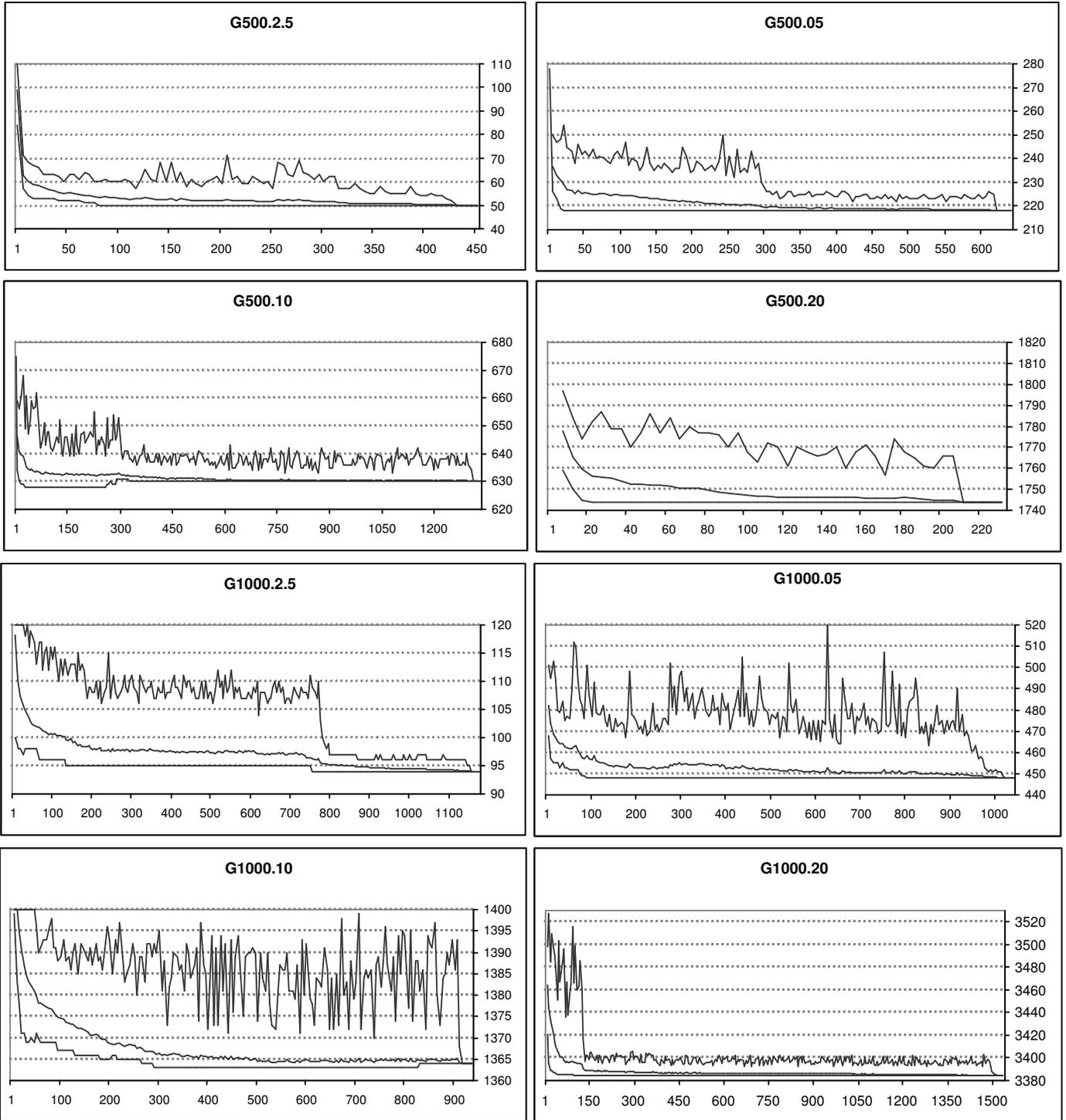
Fig. 8.   Maximum, average, and minimum solution values over the population versus generation number for *Gn.d* graphs
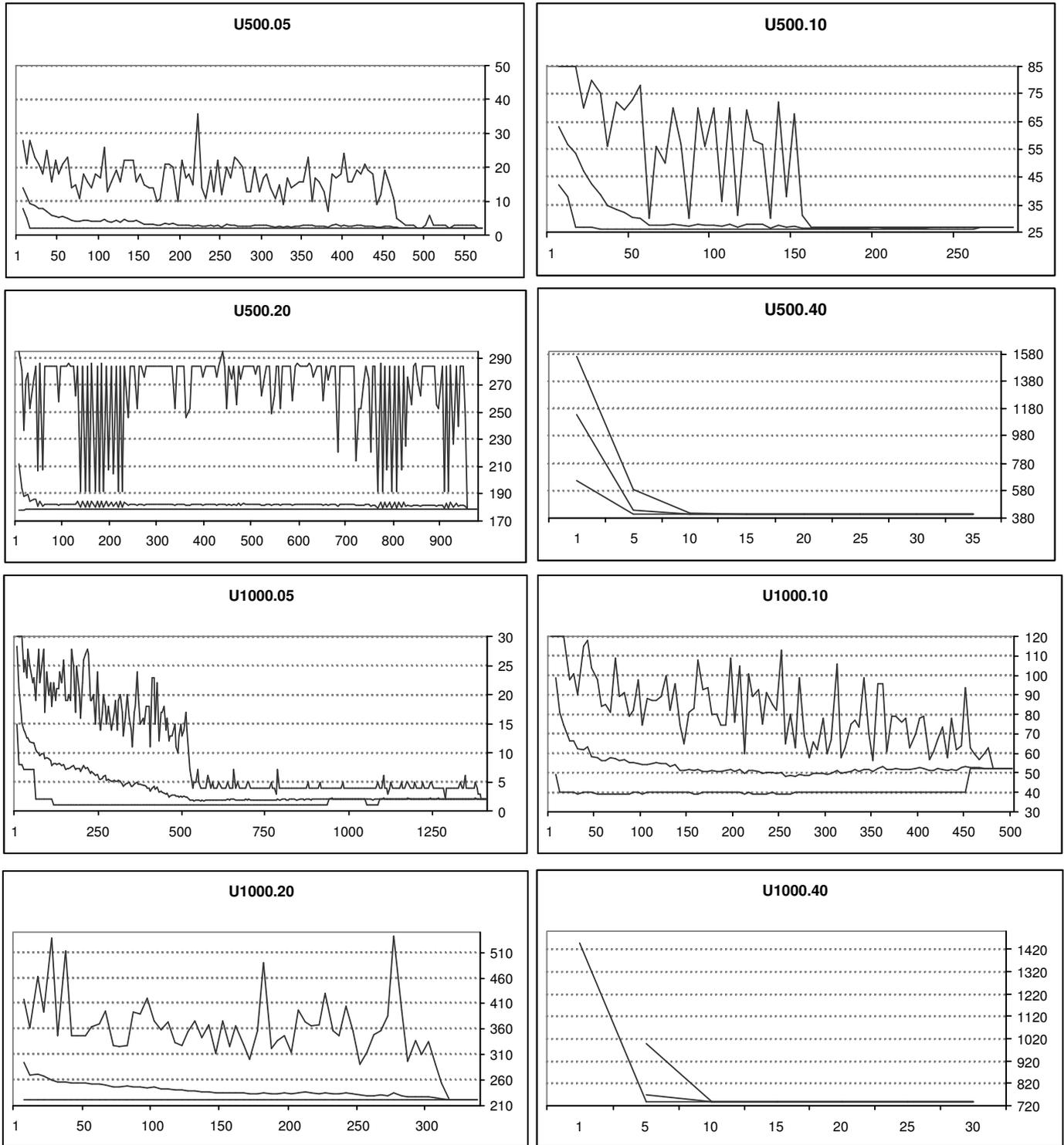
Fig. 9. Maximum, average, and minimum solution values over the population versus generation number for *Un.d* graphs

REFERENCES

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

[2] Alpert, C. and Kahng, A., "Recent Directions in Netlist Partitioning: A Survey," *Integration-The VLSI J.*, vol. 19, no. 1-2, pp. 1–81, 1995.

[3] Johannes, F.M., "Partitioning of VLSI Circuits and Systems," in *Proc. Design Automation Conf.*, 1996, pp. 83–87.

[4] Fox, G.C., "A Review of Automatic Load Balancing an Decomposition Methods for the Hypercube," in *Numerical Algorithms for Modern Parallel Computer Architectures*, Schultz, M., Ed. Springer-Verlag, 1988, pp. 63–76.

[5] Sanchis, L., "Multiple-Way Network Partitioning," *IEEE Transactions on Computers*, vol. 38, pp. 62–81, 1989.

[6] Simon, H.D., "Partitioning of Unstructured Problems for Parallel Processing," *Computing Systems in Eng.*, vol. 2, pp. 135–148, 1991.

[7] Gilbert, J.R. and Miller, G. L. and Temg, S.H., "Geometric Mesh Partitioning: Implementation and Experiments," in *Proc. Internationall Parallel Processing Symp. (IPPS'95)*, 1995.

[8] Hendrickson, B. and Leland, R., "An Improved Spectral Graph Partitioning Algorithm for Mapping Parallel Computations," *SIAM J. Scientific Computing*, vol. 16, no. 2, pp. 452–469, 1995.

[9] Ding, C. and Xiaofeng, H. and Hongyuan, Z. and Ming, G. and Simon A., "Min-Max Cut Algorithm for Graph Partitioning and Data Clustering," in *Proc. IEEE International Conf. Data Mining*, 2001, pp. 107–114.

[10] Dunlop, A. and Kernighan, B., "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.

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

[12] Kernighan, B.W. and Lin, S., "An Efficient Heuristic Procedure for Partition Graphs," *Bell Systems Technical J.*, vol. 49, pp. 291–307, February 1970.

[13] Fiduccia, C. and Mattheyses, R., "A Linear Time Heuristics for Improving Network Partitions," in *Proc. 29th ACM/IEEE Design Automation Conf.*, 1982, pp. 175–181.

[14] Garey, M. D. and Johnson, D. S. and Stockmeyer, L., "Some Simplified NP-Complete Graph Problems," *Theoretical Computer Science*, vol. 1, pp. 237–267, 1976.

[15] Johnson, D.S. and Aragon, C.R. and McGeoch, L.A. and Schevon, C., "Optimization by Simulated Annealing: An Experimental Evaluation; Part I, Graph Partitioning," *Operations Research*, vol. 37, pp. 865–892, 1989.

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

[17] Rolland, E. and Pirkul, H. and Glover, F. , "Tabu Search for Graph Partitioning," *Annals of Operational Research*, vol. 63, pp. 209–232, 1996.

[18] Battiti, R. and Bertossi, A., "Greedy, Prohibition and Reactive Heuristics for Graph Partitioning," *IEEE Transactions on Computers*, vol. 48, pp. 361–385, 1999.

[19] Bui, T.N. and Moon, B.R., "Genetic Algorithm and Graph Partitioning," *IEEE Transactions on Computers*, vol. 45, pp. 841–855, 1996.

[20] Steenbeek, A.G. and Marchiori, E. and Eiben, A. E., "Finding Balanced Graph Bipartitions Using a Hybrid Genetic Algorithm," in *Proc. of the IEEE International Conf. on Evolutionary Computation ICEC'98*, 1998, pp. 90–95.

[21] Laguna, M. and Feo, T.A. and Elrold, H.C., "Adaptive Search Procedure for the Two-Partition Problem," *Operations Research*, vol. 42, pp. 677–687, 1994.

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

[23] Karypis, G. and Aggarwal, R. and Kumar, V. and Shekhar, S., "Multilevel Hypergraph Partitioning: Application in VLSI Domain," in *Proc. of Design Automation Conf.*, 1997, pp. 526–529.

[24] Alpert, C.J. and Huang, J.-H. and Kahng, A.B. , "Multilevel Circuit Partitioning," in *Proc. Design Automation Conf.*, 1997, pp. 530–533.

[25] Saab, Y.G., "A New 2-Way Multi-Level Partitioning Algorithm," *VLSI Design J.*, vol. 11, no. 3, pp. 301–310, 2000.

[26] Saab,Y.G., "An Effective Multilevel Algorithm for Bisecting Graphs and Hypergraphs," *IEEE Transactions on Computers*, vol. 53, no. 641-652, 2004.

[27] Walshaw, Chris, "Multilevel Refinement for Combinatorial Optimisation Problems," *Annals of Operations Research*, no. 131, p. 325372, 2004.

[28] Battiti, R. and Bertossi, A., "Differential Greedy for the 0-1 Equicut Problem," in *Proc. of the DIMACS workshop on Network Design; Connectivity and Facilities Location*, Du, D.Z. and Pardalos, P.M., Ed. AMS, 1997.

[29] Barake, M. Chardaire, P. and McKeown, G. P., "The PROBE Metaheuristic for the Multiconstraint Knapsack Problem," in *Metaheuritics*, Resende, M. G. C. and de Sousa, J. P., Ed. Springer, 2004, pp. 19–36.

[30] Zbigniew Michalwicz, *Genetic Algorithms + Data Structures = Evolution Programs*. Springer, 1996.

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

[32] Duff, I.S. and Grimes, R.G. and Lewis, J.G., "Sparse Matrix Test Problems," *ACM Trans. Math. Software*, vol. 15, no. 1, pp. 1–14, 1989.

[33] Soper, A. J. and Walshaw, C. and Cross, M., "A Combined Evolutionary Search and Multilevel Optimisation Approach to Graph Partitioning," *Journal of Global Optimization*, vol. 29, no. 2, pp. 225–241, 2004.

[34] Jones, C., "Vertex and Edge Partitions of Graphs," Ph.D. dissertation, Pennsylvania State University, Pa., 1992.

Pierre Chardaire Biography text here.

PLACE
PHOTO
HERE
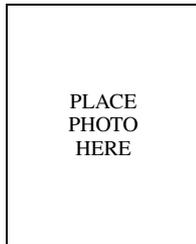
Musbah Barake Biography text here.

PLACE
PHOTO
HERE
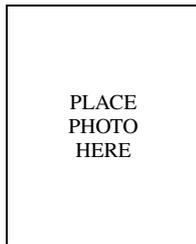
Geoff P. McKeown Biography text here.

PLACE
PHOTO
HERE