# Genetic Algorithm and Graph Partitioning

Thang Nguyen Bui and Byung Ro Moon

**Abstract**—Hybrid genetic algorithms (GAs) for the graph partitioning problem are described. The algorithms include a fast local improvement heuristic. One of the novel features of these algorithms is the *schema preprocessing* phase that improves GAs' space searching capability, which in turn improves the performance of GAs. Experimental tests on graph problems with published solutions showed that the new genetic algorithms performed comparable to or better than the multistart Kernighan-Lin algorithm and the simulated annealing algorithm. Analyses of some special classes of graphs are also provided showing the usefulness of schema preprocessing and supporting the experimental results.

**Index Terms**—Genetic algorithm, graph bisection, graph partitioning, hybrid genetic algorithm, hyperplane synthesis, multiway partitioning, schema preprocessing.

## 1 Introduction

Given a graph $G = (V, E)$ on $n$ vertices, where $V$ is the set of vertices and $E$ is the set of edges, a balanced *k-way partition* is a partitioning of the vertex set $V$ into $k$ disjoint subsets where the difference of cardinalities between the largest subset and the smallest one is at most one. Although the term $k$-way partition is also used for unbalanced partitions, we consider only balanced partition in this paper, as defined above. Hereafter, unless otherwise noted $k$-way partition means balanced $k$-way partition. The *cut size* of a partition is defined to be the number of edges whose endpoints are in different subsets of the partition. The *k-way partitioning problem* is the problem of finding a $k$-way partition with the minimum cut size. A two-way partition is often called a *bisection*. One popular approach for solving the $k$-way partitioning problem is to find bisection recursively. Thus, most existing algorithms for the partitioning problem are for the bisection problem. This paper first describes an algorithm for the bisection problem and later generalizes it for the $k$-way partitioning problem.

The $k$-way partitioning problem as well as the graph bisection problem arise in various areas of computer science, such as sparse matrix factorization, VLSI circuit placement, and network partitioning problems. The graph bisection problem has been studied extensively in the past [1], [2], [3], [4], [5]. It is known that they are NP-hard for general graphs as well as for bipartite graphs [6]. For special classes of graphs, such as trees and planar graphs with $O(\log n)$ optimal cut size exact polynomial time algorithms exist [5]. Recent results show that even finding good approximation solutions for general graphs or arbitrary planar graphs is also NP-hard [7]. Approximation algorithms for the bipartitioning problem (partitioning into two bounded but not necessarily equal sets) are available, but they are not practical [8]. In practice, a number of heuristics, i.e., algorithms that seem to work well but have no performance guarantee, are used. The most popular of these heuristics are perhaps the Kernighan-Lin algorithm (KL) [9] and the simulated annealing algorithm (SA) [10]. KL 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 them. SA is a stochastic optimization algorithm which also starts with a bisection (or sometimes just an unbalanced bisection) and improves upon it using a probabilistic hill-climbing strategy. The performance of both algorithms is known to depend on the quality of the initial bisection (or partition).

An extensive study of the simulated annealing algorithm for the graph bisection problem has been done by Johnson et al. [4]. In that study, it was observed that simulated annealing on the average performed better than KL. With the use of a compaction method [2] Kernighan-Lin algorithm, however, was observed to perform better than simulated annealing with or without this compaction technique [2], [11], [12]. Several studies using genetic algorithms (GAs) for the graph partitioning problem have also been done [13], [14], [15], [16]. However, very little empirical data exists to show their performance or to compare their performance against existing popular heuristics, such as simulated annealing. In this paper, we describe a genetic algorithm for the $k$-way partitioning problem and present the results of a rather extensive experimental evaluation of our GA using the graphs from [4], and some of our own. The quality of our results are comparable to or better than those of multistart KL and SA. A novel and distinguishing feature of our GA is the schema preprocessing phase which helps improve the quality of the solutions at very little cost in time.

Throughout this paper, we assume that the graph being discussed is $G = (V, E)$ as above, and that the number of vertices is $n$, and the number of edges is $e$. For simplicity of description we further assume, without loss of generality, that $n$ is a multiple of $k$ in the $k$-way partitioning problem. The rest of this paper is organized as follows. In Section 2, we summarize the Kernighan-Lin and simulated annealing

• *T.N. Bui is with the Department of Computer Science, Pennsylvania State University, Middletown, PA 17057. E-mail: bnt@luna.hbg.psu.edu.*
• *B.R. Moon is with the Design Technology Research Center, LG Semicon Co., Ltd., Seoul, Korea 137-140. E-mail: moon@gsen.goldstar.co.kr.*

*Manuscript received Aug. 2, 1994; revised Aug. 29, 1995.*
*For information on obtaining reprints of this article, please send e-mail to: transcom@computer.org, and reference IEEECS Log Number C96114.*

algorithms and present some basic genetic algorithm terminology. In Section 3, we describe our genetic algorithm for the graph bisection problem as well as the extension to the multiway partitioning problem. A preprocessing heuristic for GA and its analysis are provided in Section 4. In Section 5, we give our experimental results and compare our results against those of the simulated annealing and Kernighan-Lin algorithms. Finally, the conclusion is given in Section 6.

## 2 Preliminaries

In this section, we first, for completeness, briefly describe the Kernighan-Lin and the simulated annealing algorithms. We give a brief description of some genetic algorithm terminology and concepts that would help in the description of our algorithm in the next section.

### 2.1 The Kernighan-Lin Graph Bisection Algorithm (KL)

The Kernighan-Lin algorithm [9] for finding a bisection of a graph is a local optimization algorithm which improves upon a given initial bisection by swapping equal-sized subsets of the bisection to create a new bisection. This process is repeated on the new bisection either for a fixed number of times or until no improvement can be obtained.

Let $(A, B)$ be a bisection of $G = (V, E)$, i.e., $A \cup B = V$, and $A \cap B = \emptyset$. For vertices $a \in A$ and $b \in B$, denote by $g(a, b)$ the reduction in the cut size of the bisection when the two vertices $a$ and $b$ are exchanged. Denote by $g_v$ the reduction when vertex $v$ is moved into the opposite side. It is easy to see that,

$$g(a, b) = g_a + g_b - 2\delta(a, b)$$

where

$$\delta(a, b) = \begin{cases} 1, & \text{if}(a, b) \in E \\ 0, & \text{otherwise.} \end{cases}$$

The pair $(a, b)$ which maximizes $g(a, b)$ is selected. Once $a$ and $b$ are selected, they are assumed to be exchanged and not considered any more for further exchange. This way a sequence of pairs $(a_1, b_1), \ldots, (a_{n/2-1}, b_{n/2-1})$ are selected ($a_i \in A$, $b_i \in B$, $i = 1, \ldots, n/2 - 1$). The algorithm then chooses a pair $(X, Y)$, $X = \{a_1, \ldots, a_k\}$ and $Y = \{b_1, \ldots, b_k\}$, such that $\sum_{i=1}^{k} g(a_i, b_i)$ is maximized. The algorithm exchanges $X$ and $Y$. This is a pass of KL. With the bisection acquired after the exchange, KL repeats the above pass until no improvement is possible.

Fig. 1 shows the KL algorithm. There are $\Theta(n)$ iterations in the loop of lines 4–9 and it takes $O(n^2)$ time to select the pair $(a, b)$, as we need to consider all $(a, b)$ pairs (line 5). Therefore, one pass of KL takes time $O(n^3)$. It has been observed by various researchers including [9] and ourselves that with few exceptions, the number of passes (the number of iterations of the outermost loop, lines 1–12) is less than 10. Therefore, it is natural to assume that the typical complexity of KL is $O(n^3)$. Clearly, an upperbound on the number of passes taken by KL is $O(e)$ (since each pass reduces the cut size by at least one and the size of the initial bisection is at most $O(e)$). Thus, the worst case complexity of KL is $O(e \cdot n^3)$. However, no better upper bound on the number of passes taken by KL is known.

```
1.  do{
2.    Compute g_a, g_b for each a ∈ A, b ∈ B;
3.    Q_A = ∅; Q_B = ∅;
4.    for i = 1 to n/2 − 1{
5.      Choose a_i ∈ A−Q_A and b_i ∈ B − Q_B such
          that g(a_i, b_i) is maximal;
6.      Q_A = Q_A ∪ {a_i}; Q_B = Q_B ∪ {b_i};
7.      for each a ∈ A − Q_A
          {g_a = g_a + 2δ(a, a_i) − 2δ(a, b_i)}
8.      for each b ∈ B − Q_B
          {g_b = g_b + 2δ(b, b_i) − 2δ(b, a_i)}
9.    }
10.   Choose k ∈ {1, ..., n/2 − 1} to maximize
        Σ_{i=1}^{k} g(a_i, b_i);
11.   Swap the subsets {a_1, ..., a_k} and
        {b_1, ..., b_k};
12. } until (there is no improvement)
```

Fig. 1. The Kernighan-Lin algorithm.

To reduce the running time, Kernighan and Lin [9] suggested considering only the two or three vertices of highest gain values in each side and select the maximum gain pair from the combinations among them. It can be easily seen that this reduces the running time of one pass of KL to $O(n^2)$. They reported that there is little degradation in the quality of the solutions when this method is used. They also reported that their implementation showed a growth rate of $O(n^{2.4})$. In Section 3.6, it will be explained that it is possible to implement KL in $\Theta(e)$ time.

### 2.2 Simulated Annealing (SA)

The Metropolis Monte Carlo integration algorithm [17] was generalized by Kirkpatrick et al. [10] to introduce a temperature schedule for efficient searching. Simulated annealing (SA) is essentially a modified version of the hill-climbing heuristic. Starting on a point in the search space, a random move is made. If this move introduces a better point, it is accepted. This is called a positive move. If it introduces a worse point, it is accepted with a decreasing probability over time. This is called a negative move (only when accepted). The probability of negative move decreases as time goes by. Without negative moves, simulated annealing is nothing but a hill-climbing method. Negative moves provide exploration capability to SA. The negative moves allow SA to escape from local optima and therefore it is possible that SA could also miss high-quality local optima. SA has been successfully used in VLSI circuit layout problems [18], [19], [20]. Johnson et al. [4] used simulated annealing in an extensive study on the graph bisection problem and showed favorable results.

### 2.3 Genetic Algorithm (GA)

A genetic algorithm starts with a set of initial solutions (*chromosomes*), called a *population*. This population then evolves into different populations for several (frequently hundreds of) iterations. At the end the algorithm returns the best member of the population as the solution to the problem. For each iteration or generation, the evolution process proceeds as follows. Two members of the popula-

tion are chosen based on some probability distribution. These two members are then combined through a *crossover* operator to produce an offspring. With a low probability, this offspring is then modified by a *mutation* operator to introduce unexplored search space to the population, enhancing the diversity of the population (the degree of difference among chromosomes in the population). The offspring is tested to see if it is suitable for the population. If it is, a *replacement* scheme is used to select a member of the population and replace it with the new offspring. We now have a new population and the evolution process is repeated until certain condition is met, for example, after a fixed number of generations. Our genetic algorithm generates only one offspring per generation. Such a genetic algorithm is called *steady-state* genetic algorithm [21], [22], as opposed to a *generational* genetic algorithm which replaces the whole population or a large subset of the population per generation. A typical structure of a steady-state genetic algorithm is given in Fig. 2. If we add a local improvement heuristic, typically after mutation, we say it is *hybridized* and a GA with this scheme is called a *hybrid* GA. We will be providing a hybrid, steady-state GA for the graph partitioning problem.

```
create initial population of fixed size;
do {
  choose parent1 and parent2 from
    population;
  offspring = crossover(parent1, parent2);
  mutation(offspring);
  if suited(offspring) then
    replace(population, offspring);
until(stopping condition);
report the best answer;
```

Fig. 2. A typical structure of steady-state genetic algorithm.

### 2.4 Terminologies of GA

Before proceeding, we introduce some terminologies of GA that are used in this paper. A *chromosome* is a sequence of gene values. Each gene has a value from an alphabet $S$, say $\{0, 1\}$. A solution is represented by a chromosome, and for graph problems the number of vertices is often the size of the chromosome (the number of genes). For example, in the graph bisection problem each vertex in the graph is represented by a gene in a chromosome and the gene has a value 0 if the corresponding vertex is on the left hand side of the bisection and it has a value 1 if the vertex is on the right hand side of the bisection. In fact, each gene representing a vertex will occupy the same location on every chromosome. A *schema* is a pattern of genes consisting of a subset of genes at certain gene positions. More formally, if $n$ is the size of the chromosome, a schema is an $n$-tuple $\langle s_1, s_2, \ldots, s_n \rangle$ where $s_i \in S \cup \{*\}$ for $i = 1, 2, \ldots, n$. In a schema, the symbol $*$ represents don't-care positions and non-$*$ symbols (called *specific symbols*) specify defining positions of the pattern and their corresponding gene values. The number of specific symbols in a schema is called the *order* of the schema. The length between the leftmost specific symbol and the rightmost specific symbol is called the *defining length* of the schema. Consider a binary encoding of length 3, then `101` and `001` are examples of chromosomes and `10*` is a schema representing the set of chromosomes having value 1 at the first position and 0 at the second position. It is a schema of order 2 and of defining length 1. A chromosome of length $n$ can also be viewed as an instance of $2^n$ schemas. For example, the chromosome `101` is an instance of $2^3 = 8$ schemas `***`, `1**`, `*0*`, `**1`, `10*`, `1*1`, `*01`, and `101`. Although a GA does not explicitly deal with schemas, schemas are implicitly handled by GA and the concept of schema is crucial in behavioral analyses of GAs. These notations will be used in Section 4 for the arguments of preprocessing.

### 2.5 Building-Block Hypothesis and Schema Disruptivity

The formal explanation of GAs' principle of working is still open. In the previous section, it was mentioned that a chromosome is an instance of $2^n$ schemas. An explicit operation on the chromosome can also be viewed as an implicit operation on the $2^n$ schemas. That is, a GA explicitly deals with chromosomes, but it implicitly deals with exponentially many schemas. Holland [23] called this *implicit parallelism* or *intrinsic parallelism*. The principle of GAs is intuitively explained by the *building block hypothesis* (see [23] and [24]). According to the building block hypothesis, a GA implicitly gives favor to low-order, high-quality schemas and, over time, it generates higher order high-quality schemas from low-order schemas via crossover. The hypothesis states that the process is the source of GAs power of space search. For a number of schemas to combine with one another by a crossover and construct a higher order schemas, all of them must be preserved through the crossover. Here, the importance of disruptivity of a schema lies. Roughly speaking, the shorter a schema's defining length is, the less likely it is to be disrupted through crossovers. There have been previous works done on the disruptivity of schemas (see [25], [26], [27]). This paper provides a heuristic to transform perceived promising schemas into shorter counterparts to increase their survival probabilities.

## 3 Genetic Algorithm for Graph Partitioning

In this section, we describe genetic algorithms for the graph partitioning problem. For simplicity of description we will first focus our attention on the graph bisection problem; we then extend our discussion to deal with the $k$-way graph partitioning problem for $k > 2$.

### 3.1 Overview of the Algorithm

In the general structure of genetic algorithms provided in Section 2.3, we devised a crossover scheme for the graph bisection problem, a replacement scheme, and a stopping criterion. Furthermore we use a hybrid type GA and provide a fast local improvement heuristic. Lastly, we add one novel step, preprocessing, which rearranges the positions of vertices on chromosomes before GA starts. Fig. 3 shows the structure of our GA for the graph bisection problem. In the figure, the parts with bold letters represent the distinct features of our algorithm.

```
preprocess;
create initial population of fixed size p;
do {
    choose parent1 and parent2 from
      population;
    offspring = crossover(parent1, parent2);
    mutation(offspring);
    local-improvement(offspring);
    replace(population, offspring);
} until (stopping condition);
report the best answer;
```

Fig. 3. The genetic algorithm for graph bisection.

In the following, we will refer to our algorithm for the graph bisection problem as the Genetic Bisection Algorithm preprocessed by Breadth First Search (BFS-GBA). We denote by GBA the version without preprocessing. Since BFS-GBA and GBA are the same except for the preprocessing, all descriptions of GBA in the following sections are also applied to BFS-GBA. To facilitate some of the discussions to come it should be noted that we ran 1,000 trials of GBA as well as BFS-GBA on each graph that we report for the bisection problem.

### 3.2 Problem Encoding

Each solution to our problem is represented by a chromosome, which is a binary string. In our problem, a chromosome corresponds to a bisection of the graph. Thus, the terms chromosome, bisection, and solution will be used interchangeably in this paper. The number of genes in the chromosome equals $n$, the number of vertices in the graph. Each gene corresponds to a vertex in the graph. A gene has value 0 if the corresponding vertex is on, say the left side of the bisection, and has value 1 otherwise. Thus, there should be an equal number of 0s and 1s in a chromosome.

### 3.3 Initialization

GBA first creates $p$ solutions at random. Without loss of generality we can assume that the number of vertices in the graph is even. Thus, the only constraint on a chromosome is that the number of 0s and that of 1s should be equal. Usually, a larger population implies a better final solution and a longer running time. We set the population size $p$ to be 50 in our algorithm.

### 3.4 Parents Selection

We assign to each solution in the population a fitness value calculated from its cut size. The fitness value $F_i$ of a solution $i$ is calculated as follows.

$$F_i = (C_w - C_i) + (C_w - C_b)/3,$$

where

- $C_w$: cut size of the worst solution in the population,
- $C_b$: cut size of the best solution in the population,
- $C_i$: cut size of solution $i$.

Each chromosome is selected as a parent with a probability that is proportional to its fitness value. Thus, from the fitness definition it is easily seen that the probability that the best chromosome is chosen is four times as high as the probability that the worst chromosome is chosen. This is a very common parent selection scheme called *proportional selection*.

### 3.5 The Crossover and Mutation Operators

Crossover and mutation are the two most important space exploration operators in GAs. A crossover operator creates a new offspring chromosome by combining parts of the two parent chromosomes. The simplest crossover operator works as follows. It randomly selects a cut point which is the same on both parent chromosomes. The cut point divides the chromosome into two disjoint parts: the left part and the right part. The left part of parent 1 is copied to the same locations of the offspring chromosome. Similarly, the right part of parent 2 is copied to the same locations of the offspring chromosome. Let this be offspring 1. We devised one more crossover operator which is the same as the above except that it copies the *complement values* of the right part of parent 2 while it copies the left part of parent 1 unchanged. Let this be offspring 2. GBA selects the better of the two offspring and passes it to the local improver. The reason for using two crossover operators is as follows. If two chromosomes are exactly (or almost exactly) the complement of each other, they represent the same (or almost the same) bisection. The first crossover operator will create a severe inconsistency in an offspring chromosome in this case, consequently that chromosome is expected to have a poor quality.

We can choose multiple cut points instead of one. In DeJong's study, multipoint crossovers degraded the performance of GA, and the degradation increased as the number of crossover points increased [28]. There also have been empirical studies which show situations in which multipoint crossovers outperform single-point crossover [22], [29], [25]. In our experiments, crossover with five cut points generated better results than any choice of fewer cut points. The following is a possible explanation for this. It is clear that the disruptivity of a five-point crossover is considerably higher than those of a single-point crossover or a two-point crossover due to fragmented chromosomes. A crossover with higher disruptivity will make it particularly difficult for GAs to converge to local optima since GAs are not so good at fine tuning around local optima [30]. Higher schema disruptivity will certainly increase this difficulty. But there seem to be situations in which a more disruptive crossover is advantageous, for example:

1) later stages of GAs' search process when the population is quite homogeneous, and
2) when the population size is rather small.

In the first case, a disruptive crossover may provide an avenue out of a potential premature convergence, and in the second case it provides the necessary sampling accuracy for complex search spaces [31] (see also [29]). Furthermore, if GA uses local improvement (hybrid GA), some degree of disruptivity can be repaired by local improvement heuristic. Highly disruptive crossover (but not randomly disruptive) gives GAs more chance to cover unexplored space. DeJong and Sarma [32] pointed out that greater disruption is more important for steady-state GAs than generational GAs, as they suffer a higher schema loss. Fig. 4 shows our crossover operators with five cut points.

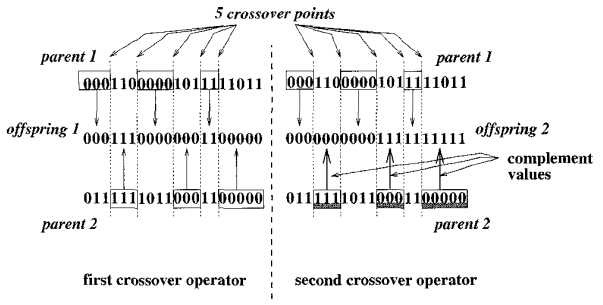



Fig. 4. Crossover operators with five cut points.

GBA applies a mutation operator as follows. $m$ positions on the chromosome are selected at random and their values are reversed (0 to 1, or 1 to 0), where $m$ is a uniform random integer variable on the interval $[0, n/100]$. As stated in Section 3.2, a chromosome must have the same number of 1s and 0s, as it represents a bisection. However, after the crossover and mutation an offspring may not have the same number of 1s and 0s. GBA counts the difference between the numbers of 1s and 0s. It then selects a random point on the chromosome and changes the required number of 1s to 0s (or 0s to 1s) starting at that point on to the right (and wrapping around if necessary). This adjustment also produces some mutation effect.

### 3.6 Local Improvement

After crossover and mutation, GBA applies a local improvement process on the offspring. Genetic algorithms are known to be not so good at fine tuning around local optima, as pointed out in the previous section [30]. Pure genetic algorithms usually take fairly many iterations but the performance is still not so desirable in many cases apparently because of this weakness. Thus, local improvement heuristics are usually applied on new offspring to fine tune them. With local improvement, GAs usually converge in considerably fewer iterations. However, local improvements themselves are costly in most cases. Since all parts other than the local improvement takes time $\Theta(n)$ or $\Theta(e)$ with few exceptions, a local improvement taking more than linear time will dominate the total running time of the GA, this is the typical problem with hybrid GAs. We reduced the running time of our hybrid GA significantly by speeding up the local improvement part. In our hybrid GA, we used a variation of the KL algorithm described in Section 2.1.

Fiduccia-Mattheyses provided a bipartition algorithm which allows unbalanced partitions to some degree and runs in $\Theta(e)$ time [33]. Fiduccia-Mattheyses algorithm chooses a vertex instead of KL's two and moves the vertex to the opposite side instead of exchanging a pair of vertices. It maintains a single gain list over all vertices, and maintains them in a bucket array with a pointer for the bucket having maximum gain vertices. This is possible because the maximum and minimum gain of a vertex is bounded by $O(n)$. They showed that the complexity is $\Theta(P)$ where $P$ is the number of pins in a circuit graph (hypergraph). If the graph does not have hyperedges (edges with more than two endpoints), it holds that $P = 2e$, so $\Theta(P) = \Theta(e)$ in this case. Even when we maintain two separate gain lists for KL, it is obvious that we can maintain them in $\Theta(e)$ time if we use the same data structure as Fiduccia-Mattheyses. Thus, we can have a KL version running in $\Theta(e)$ time.

In addition, the original KL has several passes (loop of lines 1-12 in Fig. 1). Each pass determines two equal-sized sets of vertices, one from each side of the bisection, and swap them to get a new bisection with a smaller cut size. The algorithm stops when a pass, or two consecutive passes do not produce a better bisection. The size of the sets to be exchanged in each pass can be as large as $n/2 - 1$. In our variation, we allow only one pass, furthermore, the size of the sets to be swapped is restricted to be no more than *Max-Exch-Size* (loop of lines 4-15 in Fig. 5, as opposed to loop of lines 4-9 in Fig. 1), a parameter that we have tried with different values. If *Max-Exch-Size* is chosen to be large, e.g., $n/2 - 1$, then the best bisections found by GBA are comparable to those found by the simulated annealing algorithm. However, for a number of graphs the average solution sizes are rather high. A possible reason is that a strong local optimization technique causes GBA to converge prematurely. This may be resolved by using a larger population size, but the increased running time is not welcome. On the other hand, setting *Max-Exch-Size* to be very small, say two or six, then the resulting solutions are very poor, as they are not so helpful in fine tuning. We found that setting *Max-Exch-Size* equal to $n/6 - 1$ gave the most desirable performance overall for the different graphs that we have tested. Although the two additional changes, allowing just one pass and restricting the maximum exchange size, do not decrease the asymptotic complexity, it does decrease the constant factor by about an order of 10. This is the second part of the speedup. When the number of subsets, $k$, of $k$-way partition, is greater than two, we use a similar algorithm with *Max-Exch-Size* equal to $n/(3k) - 1$.

```
1.  Compute g_a, g_b for each a ∈ A, b ∈ B;
2.  Make two gain lists of g_a s and g_b s;
3.  Q_A = ∅; Q_B = ∅;
4.  for i = 1 to Max-Exch-Size {
5.    Choose a_i ∈ A − Q_A and b_i ∈ B − Q_B such that
        g(a_i, b_i) is maximal over the two biggest
        g_a s and the two biggest g_b s;
6.    Q_A = Q_A ∪ {a_i}; Q_B = Q_B ∪ {b_i};
7.    for each a ∈ A − Q_A adjacent to a_i or b_i {
8.      g_a = g_a + 2δ(a, a_i) − 2δ(a, b_i);
9.      if g_a changed, adjust the gain list of
          side A;
10.   }
11.   for each b ∈ B − Q_B adjacent to a_i or b_i {
12.     g_b = g_b + 2δ(b, b_i) − 2δ(b, a_i);
13.     if g_b changed, adjust the gain list of
          side B;
14.   }
15. }
16. Choose k ∈ {1, ..., Max-Exch-Size}} to
      maximize Σ_{i=1}^{k} g(a_i, b_i);
17. Swap the subsets {a_1, ..., a_k} and
      {b_1, ..., b_k};
```

Fig. 5. Our variation of the Kernighan-Lin algorithm.

### 3.7 Replacement Scheme

After having generated a new offspring and trying to locally improve it, GBA replaces a member of the population with the new offspring. Invariably, we have found that the quality of the solutions depends greatly on the replacement scheme. It was observed that with a loose replacement scheme, e.g., always replace the worst member of the population with the new offspring, GBA can converge quickly at the expense of loosing diversity[1] in the population. On the other hand, with a strict replacement scheme, GBA can maintain a high diversity for a long time and consequently can get good solutions at the expense of time. We had to fix a replacement scheme which would generate good solutions in a reasonable amount of time.

Whitley and Kauth suggested a replacement scheme in which an offspring replaces the most inferior member of the population as a part of a GA framework, Genitor [21], which is still popular. In 1970, Cavicchio [34] suggested a replacement scheme, called *preselection*, in which an offspring replaces the inferior parent (only when the offspring is better) hoping to maintain population diversity. DeJong [28] suggested *crowding* in which an offspring is compared to a random subpopulation of crowding factor (CF) members and the member with the highest similarity to the offspring is replaced. In [35], the authors suggested a replacement scheme in which an offspring tries to first replace the more similar parent, measured in Hamming distance (bitwise difference), and, if it fails, then it tries to replace the other parent (replacement is done only when the offspring is better than one of the parents).

Here, we describe our experience with various replacement schemes. Using Genitor-style replacement [21], GBA converged very fast but the diversity of the population decreased significantly in the early generations. A possible reason is that this replacement scheme disregards the possibility that good schemas in the worst solutions can later blossom. Consequently, the quality of the solutions was not so desirable although it was comparable to the simulated annealing algorithm. Preselection [34] fared better than Genitor-style one, as it was able to maintain a larger diversity in the population. The diversity of the population, however, may still be relatively small, particularly if the superior parent is similar to the offspring. Since higher selection probability is given to solutions with high fitness values (i.e., small cut sizes), this scheme eventually overemphasizes the good solutions. Thus, we believe, good schemas in the inferior solution are also lost to a certain extent with this replacement scheme. The quality of the solutions improved a lot when GBA replaced the closer parent in Hamming distance (only when the offspring is better than the closer parent). The problem with this scheme is that it is very time consuming, as with large diversity GBA takes a long time to converge. In particular, this caused significant waste of generated offspring and consequently caused significant time delay. We observe that GBA generates extremely high rate of offspring waste during the latter generations. It often abandoned more than 99% of the generated offspring in the later stages of GBA.

1. The ratio of difference among solutions in population. Detailed measurement can be different from implementation to implementation.

The scheme of [35] compensates for this by combining it with preselection. Although the combined scheme performed well, relatively much wasted time was still observed in later stages of GA, as it is hard to generate an offspring better than at least one of the parents when most members of the population have very high quality.

In this paper, we combine the replacement schemes of [35] and the Genitor-style replacement scheme. We first try to replace one of the parents in the same way as in [35]; if the offspring is worse than both parents, we replace the most inferior member of the population. The rational behind this is to maintain the population diversity to the extent that not too much time is wasted. Table 1 shows the comparison between the replacement scheme of Genitor and our new scheme on GBA with over 1,000 trials on each graph. Since these are supplementary data, only a small fraction of the graphs tested are included in the table (only the first graph and the last graph in each class of graphs described in Section 5 are included). One can see that the replacement scheme of Genitor generally takes less time but the new one performs comfortably better. Both versions performed almost optimally for regular graphs and grid graphs. For all others classes of graphs, the new scheme visibly outperformed that of Genitor. Consistent results were also obtained for graphs that are not shown in the table. Note that the data in the table are statistically stable because of the large number of runs; for example, the graph U1000.40 has the largest standard deviation among all above, whose sample standard deviation (over a sample size of 1,000) is 0.064.

### 3.8 Stopping Criterion

Many genetic algorithms still run for a fixed number of generations before stopping. One of the usually better criteria for stopping is to stop when the population diversity drops below a certain threshold. The weakness of this criterion is that for our problem the best threshold of diversity is different from graph to graph, and it was not easy for GBA to find the best threshold by itself. Consequently, when we gave a threshold that is good enough for all graphs, GBA would waste a lot of time for many graphs. Setting stopping conditions differently from graph to graph does not look desirable. In our preliminary version [35], we used the number of consecutive fails to replace one of the parents as stopping criterion, which has less meaning in this combined replacement scheme. The stopping criterion that GBA used is to stop when 80% of the population is occupied by solutions with the same quality, whose chromosomes are not necessarily the same. In any case, no more than 3,000 iterations are allowed. Typical numbers of iterations are from 150 to 900.

### 3.9 Generalizing to Multiway Partition

Except for a few bisection-specific features, the genetic algorithm for multiway partition is basically the same as that for bisection. In our implementation, it is done on the same program by simply changing a parameter $k$, the number of partitions. Sections 3.3, 3.4, 3.7, and 3.8 remain exactly the same for multiway partition. In Section 3.2, the $k$-ary alphabet $\{0, 1, \ldots, k-1\}$ is used instead of the binary alphabet $\{0,$

TABLE 1
COMPARISON OF PERFORMANCE BETWEEN THE TWO REPLACEMENT SCHEMES

| Graphs | Bisection Cut Sizes Using [21]'s Replacement | | Bisection Cut Sizes Using the New Replacement | |
|---|---|---|---|---|
| | Average[1] | CPU | Average[1] | CPU |
| G500.2.5 | 55.33 | 3.86 | 54.15 | 4.72 |
| G1000.20 | 3415.14 | 50.61 | 3401.98 | 59.25 |
| U500.05 | 13.75 | 5.74 | 11.30 | 6.28 |
| U1000.40 | 752.18 | 29.58 | 742.69 | 27.88 |
| breg500.0 | 0.00 | 1.16 | 0.00 | 1.35 |
| breg5000.16 | 16.01 | 26.60 | 16.00 | 32.93 |
| cat.352 | 6.18 | 1.95 | 5.82 | 4.16 |
| rcat.5114 | 23.19 | 23.19 | 23.00 | 23.79 |
| grid100.10 | 10.00 | 0.33 | 10.00 | 0.37 |
| w-grid5000.100 | 100.72 | 74.00 | 100.00 | 86.58 |

*1. Over 1,000 runs.*

1}. In Section 3.5, the second crossover operator in Fig. 4 is not generally used since it has no advantage when $k > 2$. The adjustment after crossover for size balancing is essentially the same as in Section 3.5, except for the fact that there are more than two parts to be adjusted. The local improvement of Section 3.6 is also basically the same. Instead of the two-way Kernighan-Lin algorithm, we only have to use its extension for multiway partition [9] to make the variation. We call the generalized version the Genetic Multiway Partition Algorithm or GMPA. When schema preprocessing (to be described in the next section) is used, the algorithm is denoted by BFS-GMPA.

## 4 Schema Preprocessing

Although the performance of GBA was favorable compared to SA, it is further improved by a simple and static preprocessing heuristic. A schema is prone to be destroyed by crossovers if the locations forming the schema are scattered on a chromosome. In this section, we describe a heuristic which can preserve perceived valuable schemas and present arguments to support its usefulness. As described in Section 3.2, each gene of a chromosome represents a vertex; we can think of a chromosome as an ordered list of vertices where each vertex has value 0 or 1. The most natural vertex order to use in a chromosome is the one given by the adjacency list or the adjacency matrix. We propose a simple method of reordering the vertices on the chromosome. We first perform a breadth first search (BFS) on the input graph starting at a random vertex. The order in which the vertices are visited by the BFS is used to reorder the vertices on the chromosome. That is, position $i$ on the chromosome represents the $i$th vertex in the BFS ordering. This preprocessing method is called *BFS Reodering* and is done only once before the genetic algorithm starts.

### 4.1 The Rationale Behind Preprocessing

The definition of a schema is provided in Section 2.4. To measure the quality of a schema $H$ (a gene pattern), the most heavily used metric is $f(H)$ defined to be the average quality of chromosomes containing the schema $H$. In this paper, the smaller the cut size of a solution (chromosome), the higher is its quality. Consider a set of vertices forming a cluster (tightly connected vertices) in the graph bisection problem. Intuitively, it is expected that such a cluster should not be cut by a good bisection. Let us take Fig. 6 as an example. The four vertices in the dotted circle are tightly connected (i.e., forms a cluster) and they are expected to reside in the same side of a bisection with near optimal cut size. If the indices of the vertices are given as in the graph (a), two schemas are easily expected to have high quality (in the sense of $f(H)$): `*0**00*0` and `*1**11*1`. The defining length of both schemas is six. If they are consecutively or more closely indexed, as in the example (b), they can have shorter schemas, down to a defining length of 3. The purpose of BFS reordering is to transform perceived high-quality schemas into shorter counterparts. Clustered vertices are used as a hint for the high-quality schemas. BFS Reordering provides a way for such vertices to stay close together on the chromosome and hence the corresponding schema has a better chance of having short defining length.

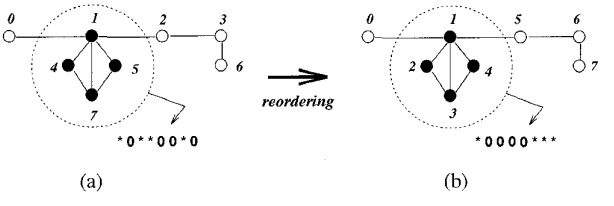



Fig. 6. Schema transformation after reordering.

More formally, let $P(t)$ represent the population at time $t$. Denote by $m(H, t)$ the expected number of chromosomes containing a schema $H$ within population $P(t)$. A bound on the expected number of chromosomes containing the schema $H$ within population $P(t + 1)$ is given by the following theorem.

SCHEMA THEOREM [23]. *In a genetic algorithm using a proportional selection and single-point crossover, the following holds for each schema H represented in P(t):*

$$m(H, t+1) \geq m(H, t)\frac{f(H, t)}{\bar{f}(t)}\left[1 - p_c \frac{\delta H}{n-1}\right], \quad (1)$$

*where $f(H, t)$ is the average fitness of the chromosomes containing the schema $H$ at time $t$, $\bar{f}(t)$ is the average fitness of all chromosomes in $P(t)$ at time $t$, $P_c$ is the crossover rate, $\delta(H)$ is the defining length of the schema $H$, and $n$ is the length of each chromosome.*

Due to the importance of this theorem, it is also called the Fundamental Theorem of Genetic Algorithms [24]. Inequality 1 says that shorter, higher quality schemas have higher survival chances as generations pass. This dependency on the defining lengths of schemas is proved to be consistent in multipoint crossovers with an odd number of cut points although they do not have a strictly linear relationship, as above [25]. BFS Reordering tends to make the defining lengths smaller for perceived high-quality schemas. Graphs with local clusters and small average degree seem to benefit most from this preprocessing technique. For example, caterpillar graphs (to be defined in Section 4.2) and geometric graphs (to be defined in Section 4.3) benefit greatly from BFS Reordering. In the following two sections, we examine the effect of BFS Reordering on those graphs. A number of different schema preprocessing heuristics are studied in [27], [36].

### 4.2 Schema Preprocessing on Caterpillar Graphs

In this section, we show that the BFS Reordering heuristic does indeed reduce the defining length of perceived high-quality schemas for the caterpillar graphs in the graph bisection problem. It should be noted that caterpillar graphs are very difficult for standard graph bisection algorithms [12].

We use $H_{n,r}$ to denote a schema of order $r$ on chromosomes of length $n$, i.e., the number of specific symbols in the schema is $r$. Let $E(\delta(H_{n,r}))$ denote the expected defining length of a schema $H_{n,r}$. If $r = \Theta(n)$, it is obvious that $E(\delta(H_{n,r})) = \Omega(n)$ since $\delta(H_{n,r}) \geq r - 1$. It is also not hard to show that $E(\delta(H_{n,2})) \simeq n/3$ if all specific symbols distribute uniformly on sufficiently large $n$ positions of a chromosome. So, $E(\delta(H_{n,r}))$, a strictly increasing function of $r$, is $\Omega(n)$ for any $r \geq 2$. It is easily seen that [37]

$$Prob[\delta(H_{n,r}) = k] = \frac{(n-k)\binom{k-1}{r-2}}{\binom{n}{r}}.$$

Using this, Goldberg et al. [38] showed that

$$E(\delta(H_{n,r})) = \frac{(r-1)}{(r+1)}(n+1). \quad (2)$$

This implies that $E(\delta(H_{n,r}))$ very quickly approaches $n$, the length of the chromosome, as $r$ increases. If we can reorder allelic positions for $E(\delta(H_{n,r}))$ to be $O(d)$, the survival probability of $H_{n,r}$ will significantly increase.

A *caterpillar graph* is a graph with sequentially connected articulation points, each having the same number of legs. Let a *unit cluster*, $C_d$, of a caterpillar graph be a subgraph consisting of an articulation point and its corresponding legs. We use $C_d$ to denote both the cluster and the corresponding schema, as there is little possibility of confusion. Fig. 7 shows a unit cluster of a caterpillar graph with each articulation point having six legs. In the case of the graph bisection problem, all the components of a unit cluster should belong to the same side, say the left side, in an optimal bisection with few exceptions. We believe that clustered vertices (in a relative sense) are usually more prone to be participants of a high-quality schema on a chromosome than an arbitrary set of vertices in most graphs. Proposition 1 and Corollary 1 say that the defining length of a schema corresponding to a unit cluster is significantly decreased by BFS Reordering. We denote BFS Reordering by $\Phi_{BF}$ from now on.

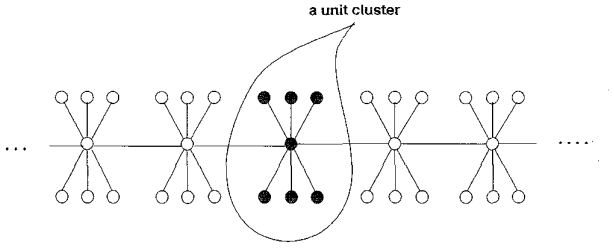



Fig. 7. A caterpillar graph with articulation points of degree 8 and a unit cluster.

PROPOSITION 1. *In a caterpillar graph with $k$ articulation points, each having $d$ legs and $k \geq \frac{3d}{2} + 1$, if we reorder the vertices on a chromosome by $\Phi_{BF}$, then the expected defining length $E(\delta(\Phi_{BF}(C_d)))$ of a schema of a unit cluster $C_d$ satisfies*

$$E(\delta(\Phi_{BF}(C_d))) \leq \frac{7d}{4} + 3.$$

PROOF. $C_d$ consists of $d + 1$ specific symbols corresponding to an articulation point and its $d$ legs. Since only articulation points can have children in the BFS tree, there are at most two articulation points in each level of the BFS tree. If there is only one articulation point in a level, the defining length of the corresponding unit cluster of the articulation point is bounded by $d + 2$ except in level 0 (in this case it is $d + 3$). If there are only two articulation points in a level, the average defining length of the corresponding unit clusters is bounded by $\frac{5d}{2} + 2$. Thus, the expected defining length satisfies

$$\begin{aligned}
&E(\delta(\Phi_{BF}(C_d))) \\
&\leq \frac{1}{\lfloor \frac{k}{2} \rfloor + 1} \sum_{i=0}^{\lfloor \frac{k}{2} \rfloor} \frac{\left(2i\left(\frac{5d}{2} + 2\right) + (k - 2i)(d+2) + 1\right)}{k} \\
&\leq \frac{1}{\lfloor \frac{k}{2} \rfloor} \sum_{i=0}^{\lfloor \frac{k}{2} \rfloor} \frac{\left(2i\left(\frac{5d}{2} + 2\right) + (k - 2i)(d+2) + 1\right)}{k} \\
&\leq \frac{7d}{4} + 2 + \frac{3d+2}{2k} \\
&\leq \frac{7d}{4} + 3.
\end{aligned}$$

□

□

COROLLARY 1. *Assuming that the number of articulation points, $k$, is such that $k \geq \frac{3d}{2} + 1$, where $d$ is the number of legs at each articulation point. If we reorder the vertices on a chromosome by $\Phi_{BF}$, then the expected defining length $E(\delta(\Phi_{BF}(C_d)))$ of a schema of a unit cluster $C_d$ satisfies:*

$$E(\delta(\Phi_{BF}(C_d))) \leq \frac{(d+2)(7d+12)}{4d(n+1)} E(\delta(C_d)).$$

PROOF. This follows immediately from (2), Proposition 1 and the fact that the order of a schema corresponding to a unit cluster is $d + 1$. □

Corollary 1 implies that BFS Reordering decreases the ratio

$$\frac{E(\delta(\Phi_{BF}(C_d)))}{E(\delta(C_d))}$$

as the number of articulation points $(= \frac{n}{(d+1)})$ in-creases. Consequently, the survival probability of $C_d$ increases significantly due to decreasing $E(\delta(C_d))$s by virtue of Section 4.1. The experimental results presented in Section 5 support the observations of this section.

### 4.3 Schema Preprocessing on Geometric Graphs

A *random geometric graph* on $n$ vertices with expected vertex degree $d$ is constructed as follows [4]. The vertices are $n$ points whose coordinates are chosen uniformly from the unit interval. There is an edge between two vertices if their Euclidean distance is $t$ or less, where $d = n\pi t^2$ is the expected vertex degree. Geometric graphs are prone to have local clusters by the way they are designed. They are one of the classes of random graphs that are believed to be most similar to actual VLSI-circuit and computer-network graphs in the sense that they tend to have local clusters. Unlike caterpillar graphs, geometric graphs do not have such simple clusters as unit clusters; thus, an analysis as in the previous section does not seem to be feasible. Instead, we examine the effect of preprocessing by plotting the defining lengths of schemas before and after BFS Reordering. Fig. 8 shows the effect of BFS Reordering on a typical sparse geometric graph of size 500 and expected vertex degree 5 (U500.05). For the plot, we divided the vertices into 50 clusters each consisting of 10 vertices by a 50-way partition using our multiway partitioning algorithm. Then, we compare the defining lengths of the schemas corresponding to every cluster before preprocessing and after preprocessing. The plot shows a significant decrease in the defining lengths after BFS Reordering.

## 5 EXPERIMENTAL RESULTS

In this section, we describe the results of experiments on a number of different graphs from [4] and on graphs of our own design. We first describe the classes of graphs that we used in the experiments. We then present the test results for the bisection problem. Finally, we present the results for the four-way partitioning problem. The results presented here reflect the changes in our algorithms since they first appeared in preliminary form in [35].

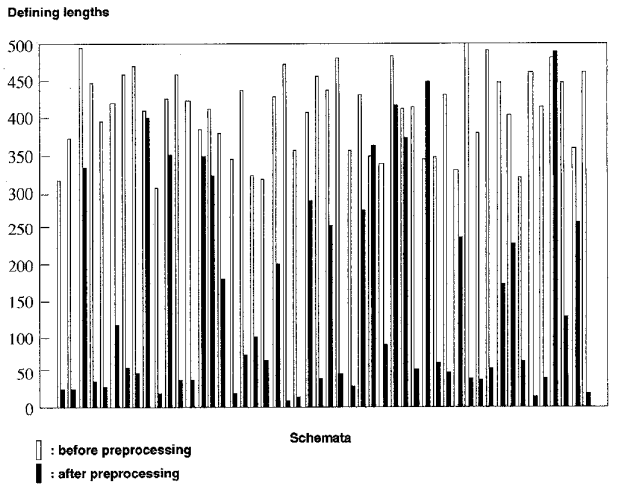



Fig. 8. The effect of preprocessing on geometric graphs.

### 5.1 The Test Graphs and Test Environment

We tested our algorithms GBA and GMPA (see Sections 3.1 and 3.9 for the naming convention of our algorithms) on a total of 40 graphs consisting of

1) 16 graphs of size (i.e., number of vertices) at least 500 that were described in [4] (eight random graphs and eight geometric graphs), and
2) 24 graphs of our own ranging in size from 134 to 5,252 (eight random regular graphs, eight caterpillar graphs, and eight grid graphs).

The different classes of graphs that we tested our algorithms on are described below. The first two classes are from [4].

- *Gn.d*: A random graph on $n$ vertices, where an edge is placed between any two vertices with probability $p$ independent of all other edges. The probability $p$ is chosen so that the expected vertex degree, $p(n-1)$, is $d$.
- *Un.d*: A random geometric graph on $n$ vertices that lie in the unit square and whose coordinates are chosen uniformly from the unit interval. There is an edge between two vertices if their Euclidean distance is $t$ or less, where $d = n\pi t^2$ is the expected vertex degree.
- breg.*n.b*: A random regular graph on $n$ vertices each of which has degree 3, and the optimal bisection size is $b$ with probability $1 - o(1)$, see [2] for more details on how such a graph is constructed. For brevity we will refer to this class of graphs as the regular graph.
- cat.*n*: A caterpillar graph on $n$ vertices, with each vertex having six legs. A caterpillar graph can be constructed as follows. Start with a straight line, called the spine, i.e., a graph in which every vertex has degree 2 except the two outermost vertices. For each vertex on the spine we add six legs. That is, for each vertex on the spine introduce six new vertices which are then connected only to that vertex on the spine by paths of length 1. With an even number of vertices on the spine it is easily seen that the optimal bisection size is 1. We use rcat.*n* to indicate a caterpillar graph with each vertex on the spine having $\sqrt{n}$ legs. All of our caterpillar graphs have optimal bisection size of 1.

- grid.*n.b*: A grid graph on $n$ vertices and whose optimal bisection size is known to be $b$. We use w-grid.*n.b* to denote the same grid but the boundaries are wrapped around.

We also tested our algorithms on smaller size graphs that were used in [4], but did not include the results in the paper, as they turned out to be too easy for both versions of GBAs. To prevent biased input, which may make the problem easier, the indices of the vertices were randomly permuted for all regular, caterpillar, and grid graphs. After considering the machines' difference in speed, our algorithms GBA and BFS-GBA overall took comparable time to that of the simulated annealing algorithm implemented in [4].[2] All programs were written in C and run on a Sun SPARC IPX.

### 5.2 Results for the Bisection Problem

We performed 1,000 runs of Genetic Bisection Algorithm (GBA) and GBA, with Breadth-First Search preprocessing (BFS-GBA) on each of the 40 test graphs. Johnson et al. [4] had 1,000 runs of the simulated annealing (SA) algorithm for each of the random graphs *Gn.d* and the geometric graphs *Un.d*. Table 2 shows the relative performance of SA, Multi-Start Kernighan-Lin (KL), GBA, and BFS-GBA on graphs provided by [4]. Table 3 shows the relative performance of Multi-Start KL, GBA, and BFS-GBA on our own graphs. Figs. 9 through 13 graphically draw the percentages above best known (or optima) of the average results of the four algorithms. The figures on the vertical axes are the percentages above best known (or optima). Since each average result was obtained from 1,000 runs, the confidence intervals of group averages are quite narrow, and thus, their inclusion would significantly impair the clarity of the plots.

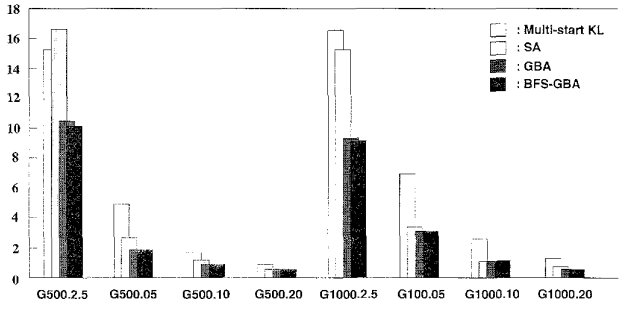



Fig. 9. Percentages of average results above best known on random graphs.

2. VAX11/750 was used in [4] with floating point accelerator, while we used a Sun SPARC IPX. The Dhrystone benchmark test data shows 1,091 Dhrystones for the VAX11/750 on VMS, 877 Dhrystones for the VAX11/750 on Unix 4.2bsd, and 1,562 for the VAX11/780 on Unix 5.2 [39]. We got 39,000 Dhrystones for the Sun SPARC IPX we used. It is listed that the SPARC Station 1 is 5.0 to 11.1 times faster than the VAX11/780 [40]. From these figures, we assume that the Sun SPARC IPX (approximately 1.75 times faster than the Sun SPARC SLC, a low model of Sun SPARC 1 line, measured by GBA) is approximately 30 times faster than the VAX11/750.

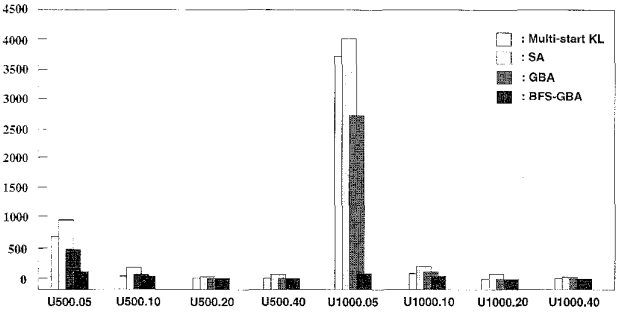



Fig. 10. Percentages of average results above best known on geometric graphs.

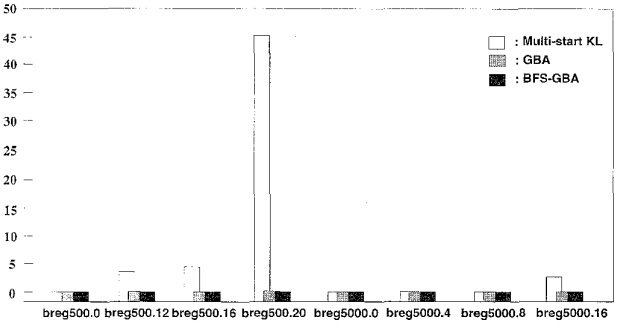



Fig. 11. Percentages of average results above optima on regular graphs.

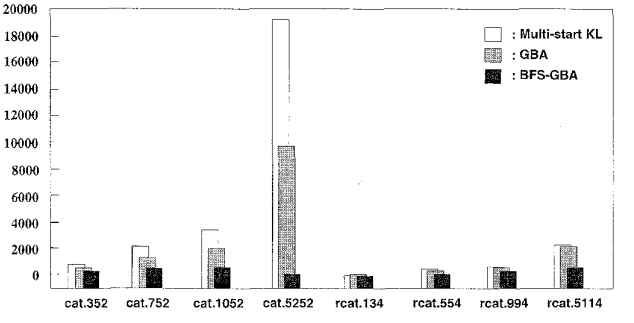



Fig. 12. Percentages of average results above optima on caterpillar graphs.

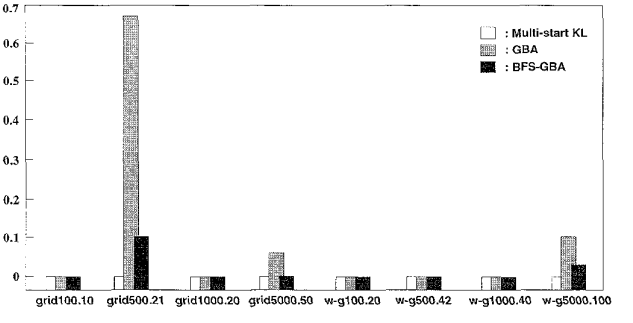



Fig. 13. Percentages of average results above optima on grid graphs.

TABLE 2
BISECTION CUT SIZES ON THE GRAPHS OF [4], WITH 1,000 TRIALS FOR EACH GRAPH

| Graphs | Cut Sizes By SA [4] | | | Cut Sizes By Multi-Start KL | | | Cut Sizes By GBA | | | Cut Sizes By BFS-GBA | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Best | Average | CPU[1] | Best[2] | Average[3] | CPU[4] | Best | Average | CPU[5] | Best | Average | CPU[5] |
| G500.2.5 | 52 | 57.20 | 379.8 | 52 | 56.51 | 5.10 | **49** | 54.15 | 4.72 | **49** | **53.97** | 5.96 |
| G500.05 | 219 | 223.82 | 308.9 | 220 | 228.62 | 7.50 | **218** | **222.11** | 6.63 | **218** | 222.13 | 8.09 |
| G500.10 | 628 | 633.65 | 341.5 | 627 | 636.73 | 12.84 | **626** | 631.61 | 11.03 | **626** | **631.46** | 11.71 |
| G500.20 | 1744 | 1752.72 | 432.9 | 1744 | 1758.30 | 28.03 | **1744** | 1752.52 | 21.29 | **1744** | **1752.51** | 21.60 |
| G1000.2.5 | 102 | 109.55 | 729.9 | 101 | 110.73 | 13.23 | 96 | 103.86 | 13.22 | **95** | **103.61** | 16.83 |
| G1000.05 | 451 | 460.02 | 661.2 | 457 | 475.94 | 17.49 | **445** | 458.71 | 20.06 | 447 | **458.55** | 23.65 |
| G1000.10 | 1367 | 1376.57 | 734.5 | 1376 | 1396.92 | 36.00 | 1363 | 1376.87 | 33.83 | **1362** | **1376.37** | 37.05 |
| G1000.20 | 3389 | 3402.56 | 853.7 | 3390 | 3427.07 | 64.20 | **3382** | 3401.98 | 59.25 | **3382** | **3401.74** | 62.25 |
| U500.05 | 4 | 21.10 | 293.3 | 5 | 15.67 | 6.46 | **2** | 11.30 | 6.28 | **2** | **3.65** | 7.54 |
| U500.10 | **26** | 65.80 | 306.3 | **26** | **31.84** | 11.60 | **26** | 38.76 | 6.94 | **26** | 32.68 | 9.59 |
| U500.20 | **178** | 247.00 | 287.2 | **178** | **178.04** | 20.72 | **178** | 182.54 | 8.32 | **178** | 179.58 | 11.50 |
| U500.40 | **412** | 680.30 | 209.9 | **412** | **412.00** | 35.36 | **412** | 412.70 | 8.05 | **412** | 412.23 | 9.92 |
| U1000.05 | 3 | 41.20 | 539.3 | 15 | 38.12 | 16.40 | 8 | 28.31 | 19.45 | **1** | **1.78** | 17.58 |
| U1000.10 | **39** | 120.40 | 563.7 | **39** | 68.41 | 30.69 | **39** | 72.79 | 20.33 | **39** | **55.78** | 30.89 |
| U1000.20 | **222** | 355.30 | 548.7 | **222** | **226.40** | 53.94 | **222** | 239.60 | 23.42 | **222** | 231.62 | 32.97 |
| U1000.40 | **737** | 963.80 | 1038.2 | **737** | **737.00** | 80.44 | **737** | 742.69 | 27.88 | **737** | 738.10 | 36.99 |

1. *CPU seconds on VAX11/750.*
2. *The best of 65,000 trials of KL.*
3. *Average of 1,000 values, each of which is the best of 65 runs of KL.*
4. *Total CPU seconds for 65 runs of KL on Sun SPARC IPX.*
5. *CPU seconds on Sun SPARC IPX.*

For random graphs where SA produced average results quite close to the best known, GBA managed to lead SA in average results. GBA led SA more comfortably in best results: It found seven new best results among the eight random graphs. For geometric graphs where SA performed not so well, GBA significantly outperformed SA in average results. GBA found one new best result among the eight geometric graphs (BFS-GBA found two new best results). But GBA could not find one best solution that SA found. The results of SA are the ones calculated from their given percentages above best known. We had to do this since a percentage above the best known had little meaning when we found a new best. Regular graphs and grid graphs turned out to be very easy graphs for GBA. GBA produced the optima as average results for most of them. For the caterpillar graphs, GBA showed results that are far from the optimal solutions. These were dramatically improved by BFS-GBA, as to be shown next.

Although BFS Reordering is a simple heuristic, the improvement resulted from this heuristic is dramatic. As mentioned above, GBA could not find the best result of one geometric graph (U1000.05) found by SA. BFS-GBA found the best known solution and further found a new best result. The best results on the graph U1000.05 were 3, 8, and 1, by SA, GBA, and BFS-GBA, respectively. The average results were 41.20, 28.31, and 1.78, respectively. As mentioned above, results produced by GBA for our caterpillar graphs were far from optima. BFS-GBA, however, dramatically improved the average results to near optimal, and found the optima for all of them. In summary, BFS-GBA worked favorably for all of the followings: random graphs, geometric graphs, regular graphs, caterpillar graphs with articulation point degree 8 or $\sqrt{N}$, grid graphs, and wrapped-around grid graphs.

A comment about the comparison with the traditional champion, Kernighan-Lin algorithm (KL), is in order. KL took roughly 1/100 of the time for SA [4]. KL, however, outperformed SA for the four densest geometric graphs in spite of its significantly smaller running time. When the bests of five runs of SA were compared with the bests of multiple-start KL allowing similar time, KL substantially outperformed SA for all geometric graphs. KL, however, was outclassed by SA for most random graphs. Our implementation of KL shows consistent performance with the implementation in [4]. BFS-GBA, our best version for bisection, takes about 53 times longer than KL does (average over all 40 test graphs). This figure will be consistent with any implementation of KL since GBA (BFS-GBA also) uses a variation of KL mentioned in Section 3.6. It took 35 times longer than KL did for regular and grid graphs, and it took 65 times longer than KL did for the other three classes of graphs. To compensate for the difference in the running time of the algorithms we run KL with many different initial bisections and return the best. Specifically, we denote by Multi-Start KL the algorithm that tries 65 runs of KL (on 65 different initial bisections) and produces the best as its final solution. Thus, the 1,000 trials of Multi-Start KL for each graph in Table 2 and Table 3 means a total of 65,000 trials of KL for each graph. Therefore, for each graph the value in the "Best" column under Multi-Start KL represents

TABLE 3
BISECTION CUT SIZES ON OUR OWN GRAPHS, WITH 1,000 TRIALS FOR EACH GRAPH

| Graphs | Optima | Cut Sizes by SA | Cut Sizes By Multi-Start KL | | | Cut Sizes By GBA | | | Cut Sizes By BFS-GBA | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | Best[2] | Average[3] | CPU[4] | Best | Average | CPU[5] | Best | Average | CPU[5] |
| breg500.0 | 0 | NA | **0** | **0.00** | 4.56 | **0** | **0.00** | 1.35 | **0** | **0.00** | 2.21 |
| breg500.12 | 12 | NA | **12** | 12.44 | 6.15 | **12** | **12.00** | 2.39 | **12** | **12.00** | 3.82 |
| breg500.16 | 16 | NA | **16** | 16.71 | 5.78 | **16** | **16.00** | 2.65 | **16** | **16.00** | 4.02 |
| breg500.20 | 20 | NA | **20** | 29.03 | 6.26 | **20** | **20.00** | 2.80 | **20** | **20.00** | 4.24 |
| breg5000.0 | 0 | NA | **0** | **0.00** | 64.71 | **0** | **0.00** | 20.33 | **0** | **0.00** | 40.22 |
| breg5000.4 | 4 | NA | **4** | **4.00** | 73.64 | **4** | **4.00** | 23.49 | **4** | **4.00** | 43.01 |
| breg5000.8 | 8 | NA | **8** | **8.00** | 81.09 | **8** | **8.00** | 26.06 | **8** | **8.00** | 44.89 |
| breg5000.16 | 16 | NA | **16** | 16.45 | 93.51 | **16** | **16.00** | 32.93 | **16** | **16.00** | 52.69 |
| cat.352 | 1 | NA | 3 | 9.34 | 1.88 | **1** | 5.82 | 4.16 | **1** | **2.25** | 2.30 |
| cat.702 | 1 | NA | 13 | 22.44 | 3.88 | 5 | 11.95 | 8.61 | **1** | **2.43** | 6.22 |
| cat.1052 | 1 | NA | 25 | 35.19 | 6.28 | 9 | 18.58 | 13.74 | **1** | **2.44** | 9.86 |
| cat.5252 | 1 | NA | 165 | 192.98 | 34.15 | 69 | 98.39 | 48.23 | **1** | **2.63** | 66.71 |
| rcat.134 | 1 | NA | **1** | 1.61 | 0.71 | **1** | 1.98 | 0.36 | **1** | **1.35** | 0.51 |
| rcat.554 | 1 | NA | **1** | 5.51 | 2.77 | **1** | 4.68 | 2.01 | **1** | **1.99** | 3.10 |
| rcat.994 | 1 | NA | 3 | 8.37 | 4.79 | **1** | 7.30 | 3.77 | **1** | **2.14** | 6.30 |
| rcat.5114 | 1 | NA | 17 | 24.45 | 28.47 | 11 | 23.00 | 23.79 | **1** | **2.36** | 52.57 |
| grid100.10 | 10 | NA | **10** | **10.00** | 0.96 | **10** | **10.00** | 0.37 | **10** | **10.00** | 0.43 |
| grid500.21 | 21 | NA | **21** | **21.00** | 6.18 | **21** | 21.14 | 3.45 | **21** | 21.02 | 3.53 |
| grid1000.20 | 20 | NA | **20** | **20.00** | 14.89 | **20** | **20.00** | 7.59 | **20** | **20.00** | 7.47 |
| grid5000.50 | 50 | NA | **50** | **50.00** | 96.13 | **50** | 50.03 | 78.23 | **50** | **50.00** | 61.16 |
| w-grid100.20 | 20 | NA | **20** | **20.00** | 0.97 | **20** | **20.00** | 0.41 | **20** | **20.00** | 0.54 |
| w-grid500.42 | 42 | NA | **42** | **42.00** | 6.22 | **42** | 42.04 | 3.44 | **42** | **42.00** | 4.67 |
| w-grid1000.40 | 40 | NA | **40** | **40.00** | 15.31 | **40** | **40.00** | 8.53 | **40** | **40.00** | 10.57 |
| w-grid5000.100 | 100 | NA | **100** | **100.00** | 92.94 | **100** | **100.00** | 86.58 | **100** | 100.03 | 94.20 |

1. *CPU seconds on VAX11/750.*
2. *The best of 65,000 trials of KL.*
3. *Average of 1,000 values, each of which is the best of 65 runs of KL.*
4. *Total CPU seconds for 65 runs of KL on Sun SPARC IPX.*
5. *CPU seconds on Sun SPARC IPX.*

the best of 65,000 trials of KL for that graph, and the "average" column represents the average of 1,000 values, each of which is the best of 65 trials of KL for that graph. This scheme might give KL a tiny advantage on regular and grid graphs. In Tables 2 and 3, bold faced figures indicate the best results (average or best bisection size) across all algorithms for each graph.

BFS-GBA outperformed the Multi-Start KL for all random graphs. Multi-Start KL marginally led BFS-GBA for five dense geometric graphs in average results. For the sparse geometric graphs, however, BFS-GBA substantially outperformed Multi-Start KL. BFS-GBA also outperformed the Multi-Start KL for regular graphs. Multi-Start KL and BFS-GBA both performed quite well for grid graphs (they produced the optimal solutions in most cases). KL is known to perform poorly on caterpillar graphs [12], which was also observed in our implementation. BFS-GBA substantially outperformed Multi-Start KL for all caterpillar graphs. Overall, when Multi-Start KL performed better (seven graphs among 40), it led BFS-GBA by a small margin. However, when BFS-GBA performed better (23 graphs among 40), the gaps were much larger.

Note that the average cut sizes obtained by BFS-GBA on the two sparsest geometric graphs (U500.05 and U1000.05) are even better than the best cut sizes from 1,000 runs of SA and Multi-Start KL. Also, the average results of BFS-GBA are better than the best results from 1,000 runs of GBA on four of the eight caterpillar graphs and on six of the eight caterpillar graphs for the case of Multi-Start KL.

### 5.3 Results of the Four-Way Partitioning Problem

We also tested the algorithms on the $k$-way partitioning problem. In particular, we consider the four-way partitioning problem. There are two alternatives for performing four-way partitioning: direct partitioning and hierarchical partitioning. A direct four-way partitioning partitions a graph into four subsets by just one run of the genetic algorithm; a hierarchical four-way partitioning first bisects a graph, then recursively bisects each of the two subgraphs to get the final four subsets. For each version, the effect of BFS Reordering was also examined. Thus, we have four algo-

TABLE 4
BEST RESULTS OF FOUR VERSIONS ON 4-WAY PARTITIONING WITH 100 TRIALS FOR EACH GRAPH

| Graphs | 4-Way Cut Sizes of GMPA | 4-Way Cut Sizes of BFS-GMPA | 4-Way Cut Sizes of Hier. GBA | 4-Way Cut Sizes of Hier. BFG-GBA |
|---|---|---|---|---|
| G500.2.5 | **86** | 91 | 87 | 89 |
| G500.05 | **364** | 365 | 370 | 370 |
| G500.10 | **1012** | **1012** | 1023 | 1026 |
| G500.20 | 2749 | **2747** | 2774 | 2778 |
| G1000.2.5 | 179 | **169** | 170 | 171 |
| G1000.05 | 745 | **737** | 755 | 753 |
| G1000.10 | 2199 | **2193** | 2225 | 2222 |
| G1000.20 | 5357 | **5354** | 5393 | 5393 |
| U500.05 | 17 | **9** | 10 | **9** |
| U500.10 | **64** | **64** | 66 | 65 |
| U500.20 | **332** | **332** | 333 | 333 |
| U500.40 | **1020** | **1020** | 1025 | 1057 |
| U1000.05 | 41 | 7 | 27 | **6** |
| U1000.10 | 87 | **84** | 92 | 87 |
| U1000.20 | **467** | **467** | 470 | 470 |
| U1000.40 | **1493** | **1493** | 1500 | 1604 |
| breg500.0 | **68** | **68** | **68** | **68** |
| breg500.12 | **72** | **72** | **72** | **72** |
| breg500.16 | **75** | **75** | **75** | **75** |
| breg500.20 | 82 | 81 | **80** | **80** |
| breg5000.0 | **652** | 660 | 670 | 674 |
| breg5000.4 | **658** | 667 | 676 | 673 |
| breg5000.8 | **670** | 673 | 683 | 680 |
| breg5000.16 | **648** | 656 | 677 | 669 |
| cat.352 | 13 | **9** | 11 | **9** |
| cat.702 | 18 | **3** | 14 | **3** |
| cat.1052 | 31 | 10 | 28 | **9** |
| cat.5252 | 169 | 10 | 145 | **9** |
| rcat.134 | **3** | **3** | **3** | **3** |
| rcat.554 | 5 | **3** | 4 | **3** |
| rcat.994 | 10 | **3** | 7 | **3** |
| rcat.5114 | 37 | 4 | 28 | **3** |
| grid100.10 | **20** | **20** | **20** | **20** |
| grid500.21 | **47** | **47** | **47** | **47** |
| grid1000.20 | **62** | **62** | **62** | **62** |
| grid5000.50 | **150** | **150** | **150** | **150** |
| w-grid100.20 | **40** | **40** | **40** | **40** |
| w-grid500.42 | **86** | **86** | **86** | **86** |
| w-grid1000.40 | **84** | **84** | **84** | **84** |
| w-grid5000.100 | **200** | **200** | **200** | **200** |

rithms to compare with one another:

1) Genetic Multiway Partitioning Algorithm (GMPA),
2) GMPA with Breadth-First Search preprocessing (BFS-GMPA) (see Section 3.9 for our naming convention),
3) hierarchical Genetic Bisection Algorithm (GBA), and
4) hierarchical GBA with Breadth-First Search preprocessing (BFS-GBA).

The results are given in Table 4 (the best results) and Table 5 (the average results and CPU seconds).

The effect of BFS Reordering was consistent with the previous section except for one case: direct partitioning of large regular graphs. The preprocessing degraded the quality of the solutions on those graphs; this phenomenon was not observed in hierarchical partitioning. The performance of direct partitioning and hierarchical one showed a sharp contrast. For caterpillar graphs, grid graphs and the four sparsest graphs in the classes of random and geometric graphs the hierarchical versions performed better than the direct versions. Direct partitioning performed well on the remaining graphs including most of the regular graphs. All these versions of the genetic algorithm used the same parameters. With these parameter settings, hierarchical partitioning took only 25% of the time of direct partitioning on the average. One may suspect that hierarchical partitioning would perform better than direct one if hierarchical partitioning is strengthened by increasing the population size so that it will take time comparable to the direct partitioning algorithm. We increased the population size from 50 to 200 for hierarchical BFS-GBA, which then took 11% longer than BFS-GMPA did. Of the 18 graphs for which hierarchical BFS-GBA performed poorer than BFS-GMPA, hierarchical BFS-GBA now outperformed BFS-GMPA on 10 graphs. For the graphs on which hierarchical partitioning still did not perform well, it is suspected that this was because a good initial *bisection* may not necessarily be a good step to a good four-way partition.

## 6 CONCLUSIONS

We presented hybrid genetic algorithms which incorporate local improvement heuristic into them. Among the new features, two that are worth emphasizing are the fast local improvement heuristic and the preprocessing. Neither the KL nor genetic algorithm without local improvement overall performed comparably to the hybrid GA. By carefully combining them together, they become quite competitive algorithms. A fast variation of KL local optimization algo-

TABLE 5
AVERAGE RESULTS OF FOUR VERSIONS ON FOUR-WAY PARTITIONING, WITH 100 TRIALS FOR EACH GRAPH

| Graphs | 4-Way Cut Sizes of GMPA | | 4-Way Cut Sizes of BFS-GMPA | | 4-Way Cut Sizes of Hier. GBA | | 4-Way Cut Sizes of Hier. BFS-GBA | |
|---|---|---|---|---|---|---|---|---|
| | Average | CPU | Average | CPU | Average | CPU | Average | CPU |
| G500.2.5 | 94.03 | 25.46 | 94.42 | 26.83 | 93.78 | 7.82 | **93.67** | 9.20 |
| G500.05 | **372.04** | 36.22 | 373.03 | 36.94 | 377.54 | 10.47 | 377.42 | 10.98 |
| G500.10 | **1024.02** | 62.98 | 1024.15 | 62.01 | 1034.03 | 16.19 | 1033.56 | 16.29 |
| G500.20 | 2763.75 | 117.80 | **2763.34** | 123.82 | 2789.92 | 29.94 | 2789.81 | 29.90 |
| G1000.2.5 | 181.82 | 78.59 | 181.36 | 92.05 | 179.58 | 22.62 | **179.33** | 26.69 |
| G1000.05 | **759.03** | 105.46 | 759.66 | 112.96 | 766.42 | 32.69 | 766.84 | 33.56 |
| G1000.10 | **2220.42** | 179.51 | 2221.84 | 182.31 | 2243.91 | 49.14 | 2243.81 | 50.15 |
| G1000.20 | 5389.02 | 306.04 | **5388.88** | 316.67 | 5422.76 | 83.81 | 5424.25 | 84.56 |
| U500.05 | 26.20 | 38.13 | 14.21 | 51.28 | 19.39 | 10.84 | **10.72** | 12.12 |
| U500.10 | 83.83 | 45.58 | **68.94** | 50.33 | 81.33 | 10.91 | 76.01 | 13.43 |
| U500.20 | 339.99 | 55.99 | **333.45** | 64.20 | 374.01 | 12.34 | 380.49 | 16.39 |
| U500.40 | 1021.67 | 87.88 | **1020.88** | 84.30 | 1057.39 | 13.32 | 1057.00 | 14.86 |
| U1000.05 | 62.28 | 92.14 | 19.42 | 136.72 | 45.22 | 31.47 | **9.37** | 28.70 |
| U1000.10 | 146.13 | 130.09 | **105.95** | 171.20 | 130.73 | 31.01 | 110.70 | 42.01 |
| U1000.20 | 496.44 | 185.30 | **486.52** | 203.26 | 502.80 | 35.22 | 489.21 | 43.50 |
| U1000.40 | 1529.97 | 269.31 | **1508.65** | 287.47 | 1651.99 | 42.22 | 1654.29 | 49.51 |
| breg500.0 | 72.70 | 18.29 | **72.12** | 22.01 | 72.52 | 4.98 | **72.12** | 5.80 |
| breg500.12 | 75.77 | 23.26 | **74.97** | 27.45 | 75.61 | 6.61 | 75.41 | 7.46 |
| breg500.16 | 78.55 | 23.04 | **78.25** | 28.27 | 78.75 | 6.81 | 78.53 | 7.78 |
| breg500.20 | 85.03 | 22.93 | 84.66 | 27.26 | 84.01 | 7.16 | **83.65** | 7.88 |
| breg5000.0 | **685.32** | 509.48 | 690.10 | 605.68 | 700.66 | 115.65 | 700.14 | 133.99 |
| breg5000.4 | **688.69** | 546.98 | 694.02 | 653.08 | 701.37 | 121.48 | 697.12 | 144.60 |
| breg5000.8 | **694.24** | 549.55 | 700.06 | 673.89 | 705.58 | 127.29 | 701.35 | 149.49 |
| breg5000.16 | **691.01** | 587.61 | 695.40 | 699.06 | 703.76 | 138.77 | 701.39 | 159.28 |
| cat.352 | 17.61 | 10.67 | 11.18 | 16.47 | 15.17 | 3.26 | **10.40** | 3.65 |
| cat.702 | 27.59 | 29.57 | 9.25 | 51.19 | 18.84 | 8.84 | **5.15** | 10.12 |
| cat.1052 | 41.77 | 56.80 | 13.25 | 94.79 | 35.62 | 17.80 | **11.05** | 16.13 |
| cat.5252 | 221.00 | 583.09 | 16.17 | 740.91 | 171.40 | 160.93 | **11.63** | 114.98 |
| rcat.134 | 4.21 | 2.33 | 3.50 | 2.54 | 3.72 | 0.72 | **3.32** | 0.80 |
| rcat.554 | 10.59 | 11.95 | 4.86 | 23.24 | 7.68 | 3.42 | **4.16** | 4.77 |
| rcat.994 | 15.98 | 25.71 | 5.67 | 50.22 | 10.85 | 6.84 | **4.18** | 10.96 |
| rcat.5114 | 43.81 | 158.21 | 8.19 | 483.01 | 35.13 | 42.56 | **4.93** | 91.54 |
| grid100.10 | 20.08 | 1.63 | 20.04 | 1.83 | **20.00** | 0.72 | **20.00** | 0.71 |
| grid500.21 | 48.58 | 20.85 | 47.55 | 22.32 | 47.03 | 5.63 | **47.00** | 5.96 |
| grid1000.20 | 66.70 | 59.75 | 62.12 | 49.19 | 62.22 | 13.92 | **62.00** | 13.82 |
| grid5000.50 | 164.03 | 699.02 | 155.51 | 624.80 | 150.06 | 142.17 | **150.00** | 115.13 |
| w-grid100.20 | **40.00** | 2.50 | **40.00** | 2.54 | **40.00** | 0.78 | **40.00** | 0.79 |
| w-grid500.42 | 91.66 | 31.70 | 91.55 | 36.30 | **88.03** | 6.37 | 88.12 | 7.75 |
| w-grid1000.40 | 84.08 | 44.50 | 84.04 | 51.70 | **84.00** | 13.26 | **84.00** | 15.33 |
| w-grid5000.100 | 222.52 | 653.59 | 205.78 | 537.94 | 202.28 | 151.83 | **200.08** | 140.79 |

rithm made it possible for the hybrid genetic algorithms to have a very practical running time. It should be noted that the power of KL itself was weakened by the speed up, but the overall power of the GA does not seem to be hurt in our experience. More thorough examination of this trade-off would be of interest.

The preprocessing heuristic provided dramatic performance improvement on some classes of graphs: sparse geometric graphs and caterpillar graphs. It took an insignificant amount of time compared to the main genetic process. Measured by the tool *gprof*, BFS Reordering took less than 2% of the total running time. We suspect that the more locally clustered a graph is, the more it is benefited by the reordering. The preprocessing method also turns out to be applicable to other problems, such as the Traveling Salesman Problem [41]; although the preprocessing heuristics are different, the concept is the same. Note that the preprocessing of encoding scheme also biases the algorithm toward particular set of solutions, which are suspected to be perceived high-quality solutions.

The performance of two four-way partitioning approaches: direct partitioning and hierarchical partitioning seemed to partially support the common belief that using a bisection algorithm to find a multiway partition may not produce as good a result as using a direct multiway partitioning algorithm (folklore). However, there are graphs in our experiments for which this observation was not true. This discrepancy is perhaps due to the fact that a direct partitioning genetic algorithm may need to take more time to converge since it is dealing with a larger search space while in our experiments equal genetic parameters were used in both the bisection and the four-way partitioning algorithms. It should be noted that a slight change in parameter (increased population size) for the hierarchical-partitioning algorithm did result in a performance improvement. It remains open to determine which of the two approaches for multiway-partitioning is the better one.

Although many partitioning algorithms, including ours, can run on a variable number of partitions ($k$), a fixed $k$ should be given before the algorithms start. But this approach may not be suitable if we would like to find natural clusters of graphs. An interesting approach would be to have an algorithm that adaptively finds the best $k$ based on some evaluating function for partitions.

## ACKNOWLEDGMENTS

We would like to thank D.S. Johnson for providing us with

the graphs of [4].

## REFERENCES


[1] R. MacGregor, "On Partitioning a Graph: A Theoretical and Empirical Study," PhD thesis, Univ. of California, Berkeley, 1978.

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

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

[4] D.S. Johnson, C. Aragon, L. McGeoch, and C. Schevon, "Optimization by Simulated Annealing: An Experimental Evaluation, Part 1, Graph Partitioning," *Operations Research*, vol. 37, pp. 865–892, 1989.

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

[6] M. Garey and D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. San Francisco: Freeman, 1979.

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

[8] F.T. Leighton and S. Rao, "An Approximate Max-Flow Min-Cut Theorem for Uniform Multicommodity Flow Problems with Applications to Approximation Algorithms," *Proc. 29th Symp. Foundations of Computer Science*, pp. 422–431, 1988.

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

[10] S. Kirkpatrick, C.D. Gelatt Jr., and M.P. Vecchi, "Optimization by Simulated Annealing," *Science*, vol. 220, no. 4598, pp. 671–680, May 1983.

[11] T.N. Bui, C. Heigham, C. Jones, and T. Leighton, "Improving the Performance of the Kernighan-Lin and Simulated Annealing Graph Bisection Algorithms," *Proc. 26th ACM/IEEE Design Automation Conf.*, pp. 775–778, 1989.

[12] C. Jones, "Vertex and Edge Partitions of Graphs," PhD thesis, Pennsylvania State Univ., University Park, Pa., 1992.

[13] J.P. Cohoon, W.N. Martin, and D.S. Richards, "A Multi-Population Genetic Algorithm for Solving the $k$-Partition Problem on Hyper-Cubes," *Proc. Fourth Conf. Genetic Algorithms*, pp. 244–248, July 1991.

[14] G. Laszewski, "Intelligent Structural Operators for the $k$-Way Graph Partitioning Problem," *Proc. Fourth Int'l Conf. Genetic Algorithms*, pp. 45–52, July 1991.

[15] Y. Saab and V. Rao, "Stochastic Evolution: A Fast Effective Heuristic for Some Genetic Layout Problems," *Proc. 27th ACM/IEEE Design Automation Conf.*, pp. 26–31, 1990.

[16] R. Collins and D. Jefferson, "Selection in Massively Parallel Genetic Algorithms," *Proc. Fourth Int'l Conf. Genetic Algorithms*, pp. 249–256, July 1991.

[17] N. Metropolis, A. Rosenbluth, M. Rosenbluth, A. Teller, and E. Teller, "Equation of State Calculations by Fast Computing Machines," *J. Chemistry and. Physics*, vol. 21, no. 6, pp. 1,087–1,092, 1953.

[18] C. Sechen and A. Sangiovanni-Vincentelli, "Timberwolf3.2: A New Standard Cell Placement and Global Routing Package," *Proc. 23rd ACM/IEEE Design Automation Conf.*, pp. 432–439, 1986.

[19] R. Rutenbar, "Simulated Annealing Algorithms: An Overview," *IEEE Circuit and Devices Magazine*, pp. 19–26, 1989.

[20] W. Sun and C. Sechen, "Efficient and Effective Placement for Very Large Circuits," *IEEE Trans. Computer-Aided Design*, vol. 14, no. 3, pp. 349–359, 1995.

[21] D. Whitley and J. Kauth, "Genitor: A Different Genetic Algorithm," *Proc. Rocky Mountain Conf. Artificial Intelligence*, pp. 118–130, 1988.

[22] G. Syswerda, "Uniform Crossover in Genetic Algorithms," *Proc. Third Int'l Conf. Genetic Algorithms*, pp. 2–9, 1989.

[23] J. Holland, *Adaptation in Natural and Artificial Systems*. Univ. of Michigan Press, 1975.

[24] D. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, 1989.

[25] K. DeJong and W. Spears, "A Formal Analysis of the Role of Multi-Point Crossover in Genetic Algorithms," *Annals of Math. AI J.*, vol. 5, pp. 1–26, 1992.

[26] S. Forrest and M. Mitchell, "Genetic Algorithms? Some Anomalous Results and Their Explanation," *Machine Learning*, to appear.

[27] T.N. Bui and B.R. Moon, "Hyperplane Synthesis for Genetic Algorithms," *Proc. Fifth Int'l Conf. Genetic Algorithms*, pp. 102–109, July 1993.

[28] K. DeJong, "An Analysis of the Behavior of a Class of Genetic Adaptive Systems," PhD thesis, Univ. of Michigan, Ann Arbor, 1975.

[29] L. Eshelman, R. Caruana, and D. Schaffer, "Biases in the Crossover Landscape," *Proc. Third Int'l Conf. Genetic Algorithms*, pp. 10–19, 1989.

[30] J. Grefenstette, "Incorporating Problem Specific Knowledge into Genetic Algorithms," *Genetic Algorithms and Simulated Annealing*, L. Davis, ed., pp. 42–60. Morgan Kaufmann, 1987.

[31] K. DeJong and W. Spears, "An Analysis of the Interacting Roles of Population Size and Crossover in Genetic Algorithms," *Parallel Problem Solving from Nature*, Lecture Notes in Computer Science, vol. 496, pp. 38-47. Springer-Verlag, 1990.

[32] K. DeJong and J. Sarma, "Generation Gaps Revisited," *Proc. Foundations of Genetic Algorithms Workshop*, 1992.

[33] C. Fiduccia and R. Mattheyses, "A Linear Time Heuristics for Improving Network Partitions," *Proc. 19th ACM/IEEE Design Automation Conf.*, pp. 175–181, 1982.

[34] D. Cavicchio, "Adaptive Search Using Simulated Evolution," PhD thesis, Univ. of Michigan, Ann Arbor, Mich., 1970. Unpublished.

[35] T.N. Bui and B.R. Moon, "A Genetic Algorithm for a Special Class of the Quadratic Assignment Problem," *The Quadratic Assignment and Related Problems*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, vol. 16, pp. 99–116, 1994.

[36] T.N. Bui and B.R. Moon, "Analyzing Hyperplane Synthesis in Genetic Algorithms Using Clustered Schemata," *Proc. Int'l Conf. Evolutionary Computation*, Lecture Notes in Computer Science, vol. 866, pp. 108-118, Springer-Verlag, Oct. 1994.

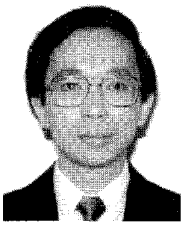
[37] D. Frantz, "Non-Linearities in Genetic Adaptive Search," PhD thesis, Univ. of Michigan, Ann Arbor, Mich., 1972.

[38] D. Goldberg, B. Korb, and K. Deb, "Messy Genetic Algorithms: Motivation, Analysis, and First Results," *Complex System*, vol. 3, pp. 493–530, 1989.

[39] B.M. Patten, private communication.

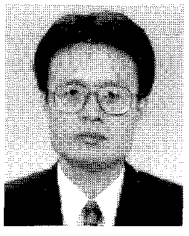
[40] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*, p. 83. Morgan Kaufmann, 1989.

[41] T.N. Bui and B.R. Moon, "A New Genetic Approach for the Traveling Salesman Problem," *Proc. IEEE Conf. Evolutionary Computation*, pp. 7–12, June 1994.

**Thang N. Bui** received BS degrees in mathematics and electrical engineering (both with honors) from Carnegie Mellon University, Pittsburgh, Pennsylvania, in 1980, and the SM and PhD degrees in computer science from the Massachusetts Institute of Technology, Cambridge, in 1983 and 1986, respectively.

Dr. Bui is an associate professor in the Computer Science Department of Pennsylvania State University. His main research interests are in the design and analysis of sequential and parallel graph algorithms. His current research interest is in genetic algorithms.



**Byung R. Moon** received the BS degree in computer science and statistics from Seoul National University (SNU), Seoul, Korea, in February 1985, the MS degree in computer science from the Korea Advanced Institute of Science and Technology (KAIST), Seoul, Korea, in February 1987, and the PhD degree from Pennsylvania State University, University Park, in December 1994.

From 1987 to 1991, Dr. Moon was an associate research engineer at the Central Laboratory, LG Electronics Co., Ltd., Seoul, Korea. During the 1991–1994 academic years, he received the Korean Government Scholarship, and during the 1992–1994 academic years, he served as a teaching assistant in the Computer Science Department, Pennsylvania State University, University Park. From November 1994 to 1995, he was a postdoctoral scholar at the VLSI CAD Lab & Commotion Lab at the University of California, Los Angeles. Since 1996, he has been a principal reserch engineer with DT Reseaarch Center, LG Semicon Co., Ltd., Seoul, Korea

His current research interests include algorithm design and analysis, genetic algorithms, VLSI circuit partitioning/placement/routing, scheduling problems, and combinatorial problems in general.