A Hybrid Genetic Search for Graph Partitioning Based on Lock Gain

Yong-Hyuk Kim and Byung-Ro Moon
School of Computer Science & Engineering, Seoul National University
Shillim-dong, Kwanak-gu, Seoul, 151-742 Korea
E-mail: yhdfly@soar.snu.ac.kr, moon@cs.snu.ac.kr

Abstract

A new hybrid genetic algorithm for graph bisection is proposed. The algorithm includes a new local optimization heuristic. Based on the traditional framework of the Kernighan-Lin algorithm, the local optimization uses a new type of gain as the primary measure for vertex movement. The new algorithm showed significant or dramatic improvement over the state-of-the-art algorithms.

1 Introduction

Let $G = (V, E)$ be an unweighted undirected graph, where V is the set of vertices and E is the set of edges. A *k-way partition* consists of k subsets of V, such that the union of the k subsets is equal to V and the subsets are mutually disjoint. In this paper, we consider only balanced partitions where the difference of cardinalities between the largest subset and the smallest one is at most one. The *cut size* of a partition is defined to be the number of edges whose endpoints are in different subsets of the partition. Formally, a balanced k-way partition P^k of the graph G is $\{C_1, C_2, \ldots, C_k\}$ such that $C_i \subset V$, $\cup_{i=1}^{k} C_i = V$, $C_i \cap C_j = \phi$ $(i \neq j)$, and $||C_i| - |C_j|| \leq 1$ for $i, j = 1, 2, \ldots, k$. The cut size of P^k is $|\{(v, w) \in E : v \in C_i, w \in C_j, i \neq j\}|$. The *k-way partitioning problem* is the problem of finding a k-way partition with minimum cut size. If the number of subsets is two, the partitioning problem is called *graph bisection* or *graph bipartitioning* problem. A general approach for a k-way partitioning problem is to find bipartitions recursively. Thus, most algorithms for graph partitioning are for the graph bisection problem. This paper also focuses on the graph bisection problem.

The graph bisection problem arises in many practical applications such as network partitioning, floor planning, VLSI circuit placement, sparse matrix factorization, parallel computing, etc. The problem has been studied extensively in the past. Since it is NP-hard for general graphs [GJ79] and even finding good approximate solutions for general graphs is also NP-hard [BJ92], practically heuristic algorithms are used. A number of heuristics for graph bisection have been proposed. These include iterative improvement algorithms [KL70] [FM82] and meta-heuristic methods such as simulated annealing (SA) [KGV83], tabu search (TS) [Glo89], large-step Markov chain (LSMC) [MOF91], and genetic algorithms (GAs) [Hol75] [Gol89]. The iterative improvement algorithms are perhaps the most basic among these heuristics. An iterative improvement algorithm is used as a heuristic in itself, as a framework for further refinement, or as a local optimization engine in hybridization with SA, TS, LSMC, GA, etc. Thus, having a good, basic iterative improvement algorithm is crucial. The Kernighan-Lin algorithm (KL) [KL70] is a representative iterative improvement heuristic. Variants of KL are discussed in [FM82], [Kri84], [DD96], etc.

A number of studies using GAs for the graph partitioning problem have been done [SR90] [CMR91] [Las91] [CJ91] [MMMR94]. However, they have limited experimental data to show their performance. Recently, Bui and Moon [BM96] used hybrid GAs in an extensive study on the graph partitioning problem. They reported superior results to multi-start KL and SA [JAMS89]. Steenbeek *et al.* [SME98] proposed a hybrid GA which exploits clusters[1]. They showed performance inbetween SA and Bui-Moon's GA.

In this paper, we suggest a hybrid genetic algorithm for graph bisection which encourages clustered movements of vertices. Like [BM96] and [SME98], effec-

[1]Subgraphs with a relatively high edge density.

tive local optimization is a major part of the GA. In most local optimization algorithms for graph bisection, poorly partitioned clusters are hard to be corrected. We devised a local optimization algorithm alleviating this problem and combined it with the power of genetic search. We did experiments on open benchmark graphs and compared the hybrid GA against state-of-the-art algorithms in the literature.

The remainder of the paper is organized as follows. In Section 2, we summarize the Kernighan-Lin algorithm. In Section 3, we introduce lock gain and describe a new local optimization algorithm. A hybrid genetic algorithm is described in Section 4. In Section 5, we present our experimental results. Finally, we make our conclusions in Section 6.

2 The Kernighan-Lin Algorithm (KL)

The Kernighan-Lin algorithm [KL70] is often considered the first reasonable heuristic for the graph bisection problem. KL proceeds in a series of *passes*. During each pass, the algorithm improves on an initial solution by swapping pairs of vertices to create a new solution. This process is repeated on the new solutions either for a fixed number of times or until no improvement can be obtained.

Let (A, B) be an initial bipartition of $G = (V, E)$. Define the *gain* g_v of a vertex v to be the cut-size reduction by moving v to the opposite set. The gain $g(a, b)$ as a result of swapping vertices $a \in A$ and $b \in B$ is as follows:

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

where

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

At the beginning of a pass, all vertices are *unlocked* or *free*, meaning that they are free to be moved. After a vertex moved, it is *locked* for the rest of the pass. KL iteratively swaps a pair of free vertices with the highest gain. The swapping process is iterated until all vertices are locked; and then the best bipartition during the pass is returned as a new solution. Another pass is then executed starting with this new solution. The algorithm terminates when one or a few passes fail to find a better solution.

The structure of the KL algorithm is given in Figure 1. A simple implementation of KL takes $O(|V|^3)$ time per pass since it requires the evaluation of $O(|V|^2)$ time to select the pair (a, b). Clearly, the number of passes is bounded by $O(|E|)$. Thus, the worst case time complexity of KL is $O(|E|\,|V|^3)$. But, in practice, significantly fewer passes (generally bounded by a constant number) are needed to complete. Therefore, it is natural to assume that the time complexity of KL is $O(|V|^3)$.

```
do {
    Compute g_a, g_b for each a ∈ A, b ∈ B;
    Q_A = φ; Q_B = φ;
    for i = 1 to |V|/2 − 1 {
        Choose a_i ∈ A − Q_A and b_i ∈ B − Q_B
          such that g(a_i, b_i) is maximal;
        Q_A = Q_A ∪ {a_i}; Q_B = Q_B ∪ {b_i};
        for each a ∈ A − Q_A
            g_a = g_a + 2δ(a, a_i) − 2δ(a, b_i);
        for each b ∈ B − Q_B
            g_b = g_b + 2δ(b, b_i) − 2δ(b, a_i);
    }
    Choose k ∈ {1, ..., |V|/2 − 1}
      to maximize Σ_{i=1}^{k} g(a_i, b_i);
    Swap the subsets { a_1, ..., a_k } and { b_1, ..., b_k };
} until (there is no improvement)
```

Figure 1: The Kernighan-Lin algorithm

To reduce the time complexity, Kernighan and Lin suggested considering only the two or three vertices with the highest gains in each set to select the max-gain pair (a, b). When this method is used, they reported that there is little degradation in the quality of the solutions. Fiduccia and Mattheyses [FM82] provided a KL-inspired algorithm which allows unbalanced partitions to some degree and reduces the time per pass to $\Theta(|E|)$. The main difference between KL and FM lies in that a new partition in FM is derived by moving a single vertex, instead of KL's pair swap. The key to the speedup in FM is the *gain bucket* data structure, which allows constant-time selection of the vertex with the highest gain and fast gain updates after each move. Even when we maintain two separate gain lists for KL, it is possible that we can maintain them in $\Theta(|E|)$ time if we use the gain bucket data structure. Bui and Moon [BM96] suggested a variation of KL with $\Theta(|E|)$ time complexity per pass. In this paper, we use this variation for KL implementation.

3 Lock-Gain Based Heuristic (LG)

3.1 Lock-Based Gain (Lock Gain)

Let (A, B) be an initial bipartition of $G = (V, E)$. For convenience, we assume that a given vertex v is in A without loss of generality. Define the *lock gain* l_v to be the gain of moving vertex v to B only with respect to the locked vertices. Formally, $l_v = |\{w \in B : (v, w) \in E \text{ and vertex } w \text{ is locked}\}| - |\{w \in A : (v, w) \in E \text{ and vertex } w \text{ is locked}\}|$. An example is given in Fig-

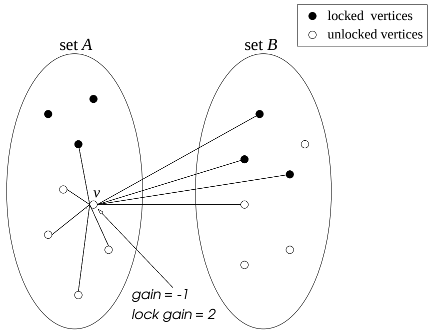

Figure 2: Lock gain in graph bisection

ure 2. In the figure, the general gain of moving v to set B is -1; but when only locked vertices are considered, the gain (lock gain) becomes 2. In particular, if all the adjacent vertices of v are locked, then l_v is equal to g_v; for the opposite case where all the adjacent vertices of v are unlocked, l_v is equal to zero.

The lock gain considers the fixed part of the gain since locked vertices do not move any more. That is, the general gain g_v consists of two types of gain: fixed gain plus changeable gain. We call them lock gain and free gain, respectively. The free gain of vertex v can change later as the free vertices move after the vertex v is locked; the motivation of the suggested algorithm is the suspicion that we have to give a preference to the unchangeable gain (lock gain) over the changeable gain (free gain).

3.2 The Algorithm

We propose a lock-gain based iterative improvement algorithm. It has the same framework as KL. The main difference with KL lies in that, instead of KL's general gains, the new algorithm uses the lock gains for vertex movements. Figure 3 shows the algorithm. It starts with an initial bipartition and improves on it through a number of passes. Let (A, B) be an initial bipartition of $G = (V, E)$. For vertices $a \in A$ and $b \in B$, denote by $l(a, b)$ the lock gain when the two vertices a and b are swapped. Then,

$$l(a, b) = l_a + l_b - 2\delta(a, b).$$

$\delta(a, b)$ was defined in Section 2. At the beginning of a pass, every vertex v is unlocked and the lock gain l_v is set to zero. We choose the pair (a, b) which maximizes $l(a, b)$ over the two biggest l_as and the two biggest l_bs. Once vertices a and b are chosen, they are assumed to be swapped and become locked. After a vertex is locked, the lock gains and general gains of its adjacent vertices are adjusted; note that the adjustments in line 10 and line 11 are different. The swapping process is iterated until a sequence of pairs $(a_1, b_1), \ldots, (a_{|V|/2-1}, b_{|V|/2-1})$ are chosen ($a_i \in A, b_i \in B, i = 1, \ldots, |V|/2 - 1$); the algorithm then finds a pair (X, Y) of subsets, $X = \{a_1, \ldots, a_k\}$ and $Y = \{b_1, \ldots, b_k\}$, such that $\sum_{i=1}^{k} g(a_i, b_i)$ is maximized, and swaps the subsets X and Y. Note that, although lock gains are used for vertex selection, the *general* gains $g(a_i, b_i)$ are used for the final decision about X and Y. Another pass is then executed with the bipartition acquired after swapping X and Y. It terminates when one or more passes fail to find an improved bipartition.

It takes $O(|V| + |E|)$ time to execute lines 2–5, 20, and 21. In the "for" loop of lines 6–19, the two adjustment loops (lines 9–13 and lines 14–18) dominate the running time. In an amortized sense, the total number of iterations of the two loops in lines 9–18 is bounded by $O(|E|)$. Therefore, if lines 12 and 17 can be done in $O(1)$ on average, the algorithm requires $O(|V| + |E|)$ time to complete one pass. In the next subsection, we will address the issue of an efficient implementation for the operation of lines 12 and 17.

3.3 Implementation

In the following, we refer the Lock-Gain based iterative improvement algorithm as LG. Like FM [FM82], we use the bucket data structure to maintain two separate lock-gain lists. This data structure allows constant-time selection of a vertex pair and fast lock-gain update after each swap. Since every lock gain is an integer in the range $[-D_{max}, D_{max}]$, where D_{max} is the maximum vertex degree in the graph, efficient management of lock gains is possible. We use the same management of buckets as that used in FM. At the beginning of a pass, all vertices are inserted into the bucket of the lock-gain value zero. Each step in the pass, a vertex is selected from a bucket and deleted from the bucket. After the vertex moves to the opposite side, the lock gains and the general gains of unlocked vertices incident to the moved vertex are updated. The update of a vertex is carried out by removing it from its lock-gain bucket and inserting it into the bucket of its new lock-gain value. If one of these vertices has a new lock gain that is larger than the current max lock gain, then the pointer to max lock gain (denoted by MaxLockGain) is replaced by this new value. If the bucket list with MaxLockGain becomes empty, then MaxLockGain is decreased until it indexes a non-empty bucket.

When a tie occurs in lock gain, we use the general gain for tie-breaking, instead of stack-based manage-

```
do {
        Compute g_a, g_b for each a ∈ A, b ∈ B;
        Set l_a and l_b to zero for each a ∈ A, b ∈ B;
        Make two lock-gain lists of l_a s and l_b s;
        Q_A = φ; Q_B = φ;
        for i = 1 to |V|/2 − 1 {
                Choose a_i ∈ A − Q_A and b_i ∈ B − Q_B such that l(a_i, b_i) is
                        maximal over the two biggest l_a s and the two biggest l_b s;
                Q_A = Q_A ∪ {a_i}; Q_B = Q_B ∪ {b_i};
                for each a ∈ A − Q_A adjacent to a_i or b_i {
                        l_a = l_a + δ(a, a_i) − δ(a, b_i);
                        g_a = g_a + 2δ(a, a_i) − 2δ(a, b_i);
                        if l_a changed then adjust the lock-gain list of side A;
                }
                for each b ∈ B − Q_B adjacent to a_i or b_i {
                        l_b = l_b + δ(b, b_i) − δ(b, a_i);
                        g_b = g_b + 2δ(b, b_i) − 2δ(b, a_i);
                        if l_b changed then adjust the lock-gain list of side B;
                }
        }
        Choose k ∈ {1, ..., |V|/2 − 1} to maximize Σ_{i=1}^{k} g(a_i, b_i);
        Swap the subsets { a_1, ..., a_k } and { b_1, ..., b_k };
} until (there is no improvement)
```

Figure 3: The lock-gain based iterative improvement algorithm

ment (LIFO strategy) in KL. If a tie occurs again in general gain (i.e., the same lock gain and the same general gain), then we apply the LIFO rule. In the implementation of our tie-breaking strategy, a removal in a bucket takes constant time but an insertion takes $O(|V|)$ due to the cost of maintaining a sorted list by general gains. The worst case time complexity of LG is $O(|V|\,|E|)$ with this implementation. We alleviated the burden for insertion by enlarging the number of buckets from $2D_{max}+1$ to $(2D_{max}+1)^2$. Here, the vertex v with lock gain l_v and general gain g_v is stored at the bucket indexed by the value $l_v(2D_{max}+1)+g_v$. In this scheme, both the removal and the insertion take $O(1)$. If D_{max} is $\Theta(|V|)$, then the cost of the algorithm is $\Theta(|V|^2)$ due to the number of buckets; if D_{max} is $O(1)$ as in most cases, the cost of the algorithm is $\Theta(|V|+|E|)$. Our experiments showed that this implementation is very efficient.

3.4 Theoretical Support

Throughout this subsection, we assume that (A, B) is the initial bisection.

Remark 1 *A* **pass** *is the process that chooses a sequence of pairs $(a_1, b_1), \ldots, (a_{|V|/2-1}, b_{|V|/2-1})$, where $a_i \in A$ and $b_i \in B$ for $i = 1, \ldots, |V|/2-1$, finds a subset pair $X = \{a_1, \ldots, a_k\}$ and $Y = \{b_1, \ldots, b_k\}$ that maximizes $\sum_{i=1}^{k} g(a_i, b_i)$, and swaps X and Y.*

Definition 1 *After applying a pass to the bisection (A, B), we have the bisection $((A-X)\cup Y, (B-Y)\cup X))$. Then, the* **real gain** *$r(a, b)$ for a pair of vertices $a \in X$ and $b \in Y$ is defined to be $g(a, b)$ in the bisection*

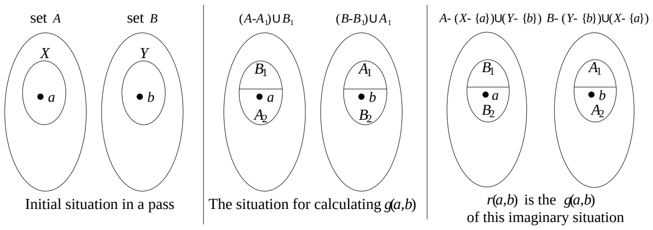

Figure 4: Real gain in graph bisection

$((A-(X-\{a\}))\cup(Y-\{b\}), (B-(Y-\{b\}))\cup(X-\{a\}))$.

Specially, if $|X| = |Y| = 1$, then $r(a, b) = g(a, b)$.

For those who are not clear about the real gain itself or its motivation, consider $X = \{a_1, \ldots, a_i, \ldots, a_k\}$ and $Y = \{b_1, \ldots, b_i, \ldots, b_k\}$ $(X \subset A, Y \subset B)$. The gain $g(a_i, b_i)$ in the middle of the pass considers the gain before the vertices $a_{i+1}, \ldots, a_k$ and $b_{i+1}, \ldots, b_k$ move. If we had known that X and Y would be the best pair for swap, it would have been better to consider the gain of (a_i, b_i) after $a_{i+1}, \ldots, a_k$ and $b_{i+1}, \ldots, b_k$ move. The real gain was devised to reflect this; the real gain $r(a_i, b_i)$ is $g(a_i, b_i)$ of the bisection $((A - \{a_1, \ldots, a_{i-1}, a_{i+1}, \ldots, a_k\}) \cup \{b_1, \ldots, b_{i-1}, b_{i+1}, \ldots, b_k\}, (B - \{b_1, \ldots, b_{i-1}, b_{i+1}, \ldots, b_k\}) \cup \{a_1, \ldots, a_{i-1}, a_{i+1}, \ldots, a_k\})$. Figure 4 shows an imaginary situation for calculating $r(a, b)$. As far as the real gain is concerned, it is not important when vertices a_i and b_i move; the only important thing is the sets X and Y. The following proposition discusses how to consider lock gains and free gains to achieve maximal real gains.

Proposition 1 *In a pass, let $A_1 \subset A$ and $B_1 \subset B$*

have been locked. If we assume that every subset pair in $A - A_1$ *and* $B - B_1$ *has the same probability to be added to the best swap set (i.e., assume* uniform probability model*), then for a pair of vertices* $a \in A - A_1$ *and* $b \in B - B_1$, $E[r(a,b)] = l(a,b)$.

Proof: By the definition of real gain, $r(a,b) =$ *fixed part* + *free part*, where the *fixed part* is $l(a,b)$ (since the gain with respect to the locked vertices does not change) and the *free part* is determined after a pass terminates. If the algorithm chooses (A_2, B_2) for the best swap set (and let the remaining sets be A_3 and B_3, i.e., $A_3 = A - (A_1 \cup A_2)$ and $B_3 = B - (B_1 \cup B_2)$), we have the bisection $(B_1 \cup B_2 \cup A_3, A_1 \cup A_2 \cup B_3)$. Assume $r(a,b) = l(a,b) + \gamma$ in this case. Now, consider the subset pair $(A_3 \cup \{a\}, B_3 \cup \{b\})$. If this pair is chosen instead of (A_2, B_2), we have the bisection $(B_1 \cup B_3 \cup \{b\} \cup (A_2 - \{a\}), A_1 \cup A_3 \cup \{a\} \cup (B_2 - \{b\}))$. In this case, $r(a,b) = l(a,b) - \gamma$. Every case has its mirror case as this example. That is, there exists a symmetry. Let the least upper bound of $r(a,b) - l(a,b)$ be t. The greatest lower bound of $r(a,b) - l(a,b)$ is then $-t$ by the symmetry. Thus, by the symmetry and the uniform-probability assumption, we also have

$$P[r(a,b) - l(a,b) = \alpha] = P[r(a,b) - l(a,b) = -\alpha]$$

where $0 \leq \alpha \leq t$.

Therefore,

$$\begin{aligned} E[r(a,b)] &= \sum_{\alpha=-t}^{t} (l(a,b) + \alpha) P[r(a,b) - l(a,b) = \alpha] \\ &= \sum_{\alpha=-t}^{t} l(a,b) P[r(a,b) - l(a,b) = \alpha] \\ &= l(a,b) \sum_{\alpha=-t}^{t} P[r(a,b) - l(a,b) = \alpha] \\ &= l(a,b). \end{aligned}$$

Q.E.D.

The uniform-probability assumption is not very general; the Proposition 1 is thus not a definite guide for iterative improvement algorithms. However, it strongly suggests that, unless otherwise clarified, one had better not consider the free gains.

4 A Hybrid Genetic Algorithm

When a genetic algorithm is hybridized with a local improvement heuristic, it is said to be a hybrid GA. Several studies about hybridization of GAs have been done [WGM94] [RB94] [LG97]. The general structure of hybrid steady-state genetic algorithms is used in our GA. In the following, we denote the framework by GBA (the Genetic Bisection Algorithm).

- *Encoding*: In this problem, a chromosome corresponds to a bipartition (A, B) of the graph $G = (V, E)$. The number of genes in a chromosome is equal to $|V|$. Each gene corresponds to a vertex in the graph. A gene has value zero if the vertex belongs to the set A; otherwise, it has value one.

- *Initialization*: GBA first creates p bipartitions at random. The only constraint on a chromosome is that the difference between the number of 0's and that of 1's should be at most one. We set the population size p to be 50 in GBA.

- *Selection*: We assign to each chromosome in the population a fitness value calculated from its cut size. GBA uses the roulette-wheel-based *proportional selection* scheme.

- *Crossover and Mutation Operators*: A crossover operator creates a new offspring by combining parts of the two parents. In our experiments, we use the crossover of [BM96] with five cut points and use no mutation. After the crossover, an offspring may not satisfy the balance requirement. It then selects a random point on the chromosome and changes the required number of 1's to 0's (or 0's to 1's) starting at that point on to the right. This adjustment produces some mutation effect.

- *Local Optimization*: After crossover and adjustment, GBA applies a local optimization on the offspring. In our hybrid GA, we use KL and LG.

- *Replacement*: After generating an offspring and applying a local optimization on it, GBA replaces a member of the population with the offspring. We use the replacement scheme of [BM96]. The offspring tries to first replace the more similar parent, measured in 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). If the offspring is worse than both parents, we replace the worst member of the population (Genitor-style replacement [WK88]).

- *Stopping Condition*: For stopping, we use the number of consecutive fails to replace one of the parents. We set the number to be 20 in GBA.

5 Experimental Results

5.1 Test Beds and Test Environment

We tested the proposed algorithm on a total of 28 graphs which consist of three groups of graphs. Twenty four of them are the same graphs used in [BM96]. They are composed of 16 graphs used in [JAMS89] (8 random graphs and 8 geometric graphs) and 8 graphs used in [BM96] (8 caterpillar graphs). They have been used in a number of other studies [BM96] [SME98] [MF98] [BB99]. The other four graphs were made by ourselves for this test. The different classes of graphs are briefly described below.

- G$n.d$: 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.
- U$n.d$: 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.
- cat.n : A caterpillar graph on n vertices, with each vertex having six legs. It is constructed by starting with a straight line (called the spine), where each vertex has degree two except the outermost vertices. 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 is a caterpillar graph with n vertices, where each vertex on the spine has $\sqrt{n}$ legs. All caterpillar graphs have an optimal cut size of 1.

All programs were written in C language and run on an Alpha PC 600 MHz with Linux operating system. They were compiled using GNU's *gcc* compiler. We performed 1,000 runs for all experiments so the confidence intervals of the experimental data are quite narrow.

5.2 Performance

Table 1 shows the performance of KL and LG. LG significantly outperformed KL for all tested graphs. In particular, LG showed dramatic performance improvement on sparse geometric graphs and caterpillar graphs. Sparse geometric graphs are prone to have local clusters by the way they are designed. Caterpillar graphs consist of unit clusters, where a *unit cluster* is a subgraph consisting of an articulation point and its corresponding legs. For these locally clustered graphs, LG performed very well. It should be noted that caterpillar graphs are known to be difficult for standard graph bisection algorithms [Jon92]. In summary, LG dominated KL for all tested graphs with particularly dramatic improvement for graphs with clusters. Although LG performed well, it seems that there is still room for further improvement; when postprocessed by KL, LG showed visible improvement for most graphs. This result is given in the last column of Table 1. We call this version Postprocessed LG or PLG.

Table 2 compares the performance of our GA with state-of-the-art genetic algorithms for graph bisection. We denote by PLG-GBA our genetic bisection algorithm using PLG as a local optimizer. KL-GBA and BFS-GBA are enhanced versions of the GAs in [BM96]. CE-GA is the GA in [SME98]. PLG-GBA and KL-GBA have the same framework except the local optimization part. BFS-GBA is the KL-GBA preprocessed with genetic reordering by breath-first search [BM96] and it performed better than KL-GBA. PLG-GBA overall significantly outperformed the others. In addition, even its average results were exactly the same as the best known or optimal solutions except for random graphs and two large sparse geometric graphs with 5,000 vertices. To the best of our knowledge, PLG-GBA is showing the best performance in the literature with respect to the benchmark graphs. Since it is a hybrid GA, the local optimization with PLG dominates the running time. We observed that the number of calls for PLG in a GA run was from 181 to 846 depending on graphs.

6 Conclusions

In this paper, we improved the performance of KL through two steps. First, it was improved by using the lock gain as the primary measure for vertex movement. Second, by combining with the framework of genetic algorithms, it showed one more dramatic improvement.

The key idea in this paper is that we used lock gains in an iterative improvement process for graph bisection. The PLG-GBA is a good example to utilize PLG. PLG can be combined with other metaheuristics such as SA [KGV83], tabu search [Glo89] [BB99], LSMC [MOF91] [HKM97], etc. We believe that PLG is a strong candidate to improve existing results in those frameworks, as it did in the framework of GA in this paper. Experimentation with respect to these methods is left for future study.

Table 1: Comparison of KL and Lock-Gain Based Heuristics

Graph	KL		LG		PLG[2]	
	Ave(Min)[1]	CPU	Ave(Min)[1]	CPU	Ave(Min)[1]	CPU
G500.2.5	64.99(53)	0.0033	**59.00**(52)	0.0047	58.50(52)	0.0067
G500.05	244.98(221)	0.0051	**236.20**(219)	0.0108	234.60(219)	0.0135
G500.10	656.46(628)	0.0105	**651.03**(627)	0.0239	648.38(626)	0.0268
G500.20	1787.48(1752)	0.0202	**1784.19**(1753)	0.0554	1779.03(1750)	0.0683
G1000.2.5	126.06(103)	0.0104	**111.87**(99)	0.0131	110.98(99)	0.0153
G1000.05	501.20(470)	0.0142	**481.23**(458)	0.0335	478.77(457)	0.0425
G1000.10	1435.09(1392)	0.0254	**1415.64**(1375)	0.0697	1412.63(1374)	0.0810
G1000.20	3482.98(3409)	0.0526	**3467.56**(3402)	0.1538	3461.27(3397)	0.1695
U500.05	38.29(12)	0.0048	**5.75**(2)	0.0072	5.64(2)	0.0084
U500.10	90.20(26)	0.0080	**30.84**(26)	0.0134	30.21(26)	0.0147
U500.20	221.01(178)	0.0141	**196.61**(178)	0.0233	193.92(178)	0.0259
U500.40	436.57(412)	0.0234	**423.98**(412)	0.0411	414.80(412)	0.0442
U1000.05	74.57(26)	0.0139	**3.89**(1)	0.0203	3.82(1)	0.0215
U1000.10	163.84(51)	0.0216	**52.23**(39)	0.0323	51.37(39)	0.0369
U1000.20	312.13(222)	0.0380	**265.87**(222)	0.0681	260.47(222)	0.0800
U1000.40	860.00(737)	0.0625	**811.56**(737)	0.1363	801.64(737)	0.1603
cat.352	20.70(7)	0.0010	**3.39**(1)	0.0020	3.39(1)	0.0025
cat.702	41.80(19)	0.0020	**3.48**(1)	0.0050	3.48(1)	0.0063
cat.1052	58.36(29)	0.0033	**3.35**(1)	0.0088	3.35(1)	0.0101
cat.5252	251.79(177)	0.0233	**3.15**(1)	0.0893	3.15(1)	0.0989
rcat.134	10.74(1)	0.0003	**2.18**(1)	0.0007	2.18(1)	0.0008
rcat.554	34.14(3)	0.0013	**3.29**(1)	0.0035	3.29(1)	0.0042
rcat.994	58.17(5)	0.0027	**2.65**(1)	0.0081	2.65(1)	0.0089
rcat.5114	190.09(21)	0.0173	**3.08**(1)	0.0619	3.08(1)	0.0693
U2000.05	162.35(75)	0.0275	**10.03**(3)	0.0560	9.95(3)	0.0572
U2000.10	346.93(71)	0.0505	**69.34**(47)	0.0976	68.30(47)	0.1079
U5000.05	433.93(251)	0.0906	**14.06**(4)	0.1991	13.96(4)	0.2181
U5000.10	956.87(469)	0.1775	**115.85**(74)	0.3568	114.73(74)	0.4026

1. 1,000 runs on Alpha PC 600 MHz.
2. PLG denotes LG postprocessed by KL.

Acknowledgments

This work was supported by Brain Korea 21 Project.

References

[BB99] R. Battiti and A. Bertossi. Greedy, prohibition, and reactive heuristics for graph partitioning. *IEEE Trans. on Computers*, 48(4):361–385, 1999.

[BJ92] T. N. Bui and C. Jones. Finding good approximate vertex and edge partitions is NP-hard. *Information Processing Letters*, 42:153–159, 1992.

[BM96] T. N. Bui and B. R. Moon. Genetic algorithm and graph partitioning. *IEEE Trans. on Computers*, 45(7):841–855, 1996.

[CJ91] R. Collins and D. Jefferson. Selection in massively parallel genetic algorithms. In *Fourth International Conference on Genetic Algorithms*, pages 249–256, 1991.

[CMR91] J. P. Cohoon, W. N. Martin, and D. S. Richards. A multi-population genetic algorithm for solving the k-partition problem on hyper-cubes. In *Fourth International Conference on Genetic Algorithms*, pages 244–248, 1991.

[DD96] S. Dutt and W. Deng. A probability-based approach to VLSI circuit partitioning. In *Design Automation Conference*, pages 100–105, June 1996.

[FM82] C. Fiduccia and R. Mattheyses. A linear time heuristics for improving network partitions. In *19th ACM/IEEE Design Automation Conference*, pages 175–181, 1982.

[GJ79] M. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* Freeman, San Francisco, 1979.

[Glo89] F. Glover. Tabu search – Part I. *ORSA Journal on Computing*, 1:190–206, 1989.

[Gol89] D. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison-Wesley, 1989.

[HKM97] I. Hong, A. B. Kahng, and B. R. Moon. Improved large-step Markov chain variants for the symmetric TSP. *Journal of Heuristics*, 3(1):63–81, 1997.

[Hol75] J. Holland. *Adaptation in Natural and Artificial Systems.* University of Michigan Press, 1975.

[JAMS89] D. S. Johnson, C. Aragon, L. McGeoch, and C. Schevon. Optimization by simulated annealing: An experimental evaluation, Part 1, graph partitioning. *Operations Research*, 37:865–892, 1989.

[Jon92] C. Jones. *Vertex and Edge Partitions of Graphs.* PhD thesis, Penn. State Univ., University Park, PA, 1992.

[KGV83] S. Kirkpatrick, C. D. Jr. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, May 1983.

[KL70] B. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *Bell Systems Technical Journal*, 49:291–307, Feb. 1970.

[Kri84] B. Krishnamurthy. An improved min-cut algorithm for partitioning VLSI networks. *IEEE Trans. on Computers*, C-33:438–446, 1984.

[Las91] G. Laszewski. Intelligent structural operators for the k-way graph partitioning problem. In *Fourth International Conference on Genetic Algorithms*, pages 45–52, July 1991.

[LG97] F.G. Lobo and D.E. Goldberg. Decision making in a hybrid genetic algorithm. In *IEEE International Conference on Evolutionary Computation*, pages 121 –125, 1997.

Table 2: Bisection Cut Sizes of Four Algorithms

Graph	Best[3] known	CE-GA [SME98] Ave[2]	CE-GA [SME98] CPU[5]	KL-GBA Ave(Min)[1]	KL-GBA CPU[4](Gen[6])	BFS-GBA Ave(Min)[1]	BFS-GBA CPU[4](Gen[6])	PLG-GBA Ave(Min)[1]	PLG-GBA CPU[4](Gen[6])
G500.2.5	49	54.1	24.9	51.66(49)	1.48(639)	51.87(50)	1.43(627)	**50.41**(49)	2.48(512)
G500.05	218	221.8	23.7	218.87(218)	2.39(753)	219.07(218)	2.48(772)	**218.04**(218)	5.60(577)
G500.10	626	631.1	27.5	627.94(626)	4.24(695)	627.85(626)	4.84(701)	**626.85**(626)	12.05(588)
G500.20	1744	1750.3	33.4	1746.21(1744)	9.54(724)	1746.12(1744)	9.40(728)	**1745.59**(1744)	29.42(634)
G1000.2.5	93	104.5	79.2	97.91(94)	4.78(856)	98.25(95)	4.99(849)	**96.23**(93)	6.81(555)
G1000.05	445	458.5	79.9	452.75(445)	8.52(981)	453.00(445)	8.95(970)	**449.49**(445)	22.89(693)
G1000.10	1362	1374.6	79.5	1367.86(1362)	15.19(1007)	1367.55(1362)	16.87(1008)	**1364.42**(1362)	45.16(846)
G1000.20	3382	3396.8	85.8	3386.21(3382)	31.62(1027)	3386.05(3382)	33.61(1025)	**3384.49**(3382)	77.90(793)
U500.05	2	2.2	13.4	6.26(2)	2.42(761)	3.16(2)	2.32(750)	**2.00**(2)	1.93(347)
U500.10	26	**26.0**	10.5	26.40(26)	2.54(501)	26.09(26)	3.13(558)	**26.00**(26)	2.42(278)
U500.20	178	**178.0**	26.3	178.03(178)	3.08(344)	**178.00**(178)	3.66(374)	**178.00**(178)	6.29(304)
U500.40	412	**412.0**	9.2	**412.00**(412)	2.54(190)	**412.00**(412)	2.64(196)	**412.00**(412)	5.38(181)
U1000.05	1	3.2	43.3	14.17(1)	8.21(1040)	1.40(1)	6.66(999)	**1.00**(1)	3.35(245)
U1000.10	39	**39.0**	20.1	46.73(39)	8.41(702)	45.04(39)	8.74(650)	**39.00**(39)	8.75(320)
U1000.20	222	225.9	37.1	222.17(222)	7.61(407)	**222.00**(222)	11.04(471)	**222.00**(222)	18.79(333)
U1000.40	737	738.2	38.1	**737.00**(737)	8.55(253)	**737.00**(737)	10.48(261)	**737.00**(737)	24.15(254)
cat.352	1	-	-	5.28(1)	0.41(440)	2.52(1)	0.39(378)	**1.00**(1)	0.42(257)
cat.702	1	-	-	12.74(5)	0.96(515)	2.74(1)	1.09(467)	**1.00**(1)	0.98(253)
cat.1052	1	-	-	21.24(9)	1.72(558)	2.83(1)	2.05(515)	**1.00**(1)	1.68(259)
cat.5252	1	-	-	119.32(61)	17.58(921)	2.90(1)	21.82(660)	**1.00**(1)	12.24(240)
rcat.134	1	-	-	1.25(1)	0.10(326)	1.01(1)	0.09(261)	**1.00**(1)	0.12(211)
rcat.554	1	-	-	3.89(1)	0.52(383)	1.78(1)	0.50(308)	**1.00**(1)	0.83(253)
rcat.994	1	-	-	6.17(1)	1.04(402)	2.27(1)	1.11(348)	**1.00**(1)	1.37(243)
rcat.5114	1	-	-	21.03(13)	7.56(432)	2.75(1)	10.35(462)	**1.00**(1)	9.79(253)
U2000.05	-	-	-	43.85(15)	15.28(1589)	8.89(4)	11.10(1268)	**3.00**(3)	15.32(385)
U2000.10	-	-	-	90.96(47)	20.11(1218)	87.75(47)	18.95(1016)	**47.00**(47)	26.66(402)
U5000.05	-	-	-	152.48(90)	69.99(2489)	24.83(12)	39.97(1521)	**4.01**(4)	64.84(477)
U5000.10	-	-	-	319.37(137)	105.02(2048)	140.85(84)	73.73(1420)	**73.28**(73)	178.22(535)

1. From 1,000 runs.
2. From 100 runs.
3. The best known from the literature so far.
4. CPU seconds on Alpha PC 600 MHz.
5. CPU seconds on SGI-O2 workstation with a 180 MHz R5000 processor.
6. Average generation from 1,000 runs.

[MF98] P. Merz and B. Freisleben. Memetic algorithms and the fitness landscape of the graph bi-partitioning problem. In *Proceedings of the 5th International Conference on Parallel Problem Solving From Nature*, 1998. *Lecture Notes in Computer Science*, 1498:765-774, Springer-Verlag.

[MMMR94] H. S. Maini, K. G. Mehrotra, C. K. Mohan, and S. Ranka. Genetic algorithms for graph partitioning and incremental graph partitioning. In *Proceedings of Supercomputing '94*, pages 449–457, 1994.

[MOF91] O. C. Martin, S. W. Otto, and E. W. Felten. Large-step Markov chains for the traveling salesman problem. *Complex Systems*, 5(3):299–326, 1991.

[RB94] J.-M. Renders and H. Bersini. Hybridizing genetic algorithms with hill-climbing methods for global optimization: Two possible ways. In *Proceedings of the First IEEE Conference on Evolutionary Computation*, pages 312–317, 1994.

[SME98] A.G. Steenbeek, E. Marchiori, and A. E. Eiben. Finding balanced graph bi-partitions using a hybrid genetic algorithm. In *IEEE International Conference on Evolutionary Computation*, pages 90–95, 1998.

[SR90] Y. Saab and V. Rao. Stochastic evolution: A fast effective heuristic for some genetic layout problems. In *27th ACM/IEEE Design Automation Conference*, pages 26–31, 1990.

[WGM94] D. Whitley, V. Gordon, and K. Mathias. Larmarckian evolution, the baldwin effect and function optimization. In *International Conference on Evolutionary Computation*, Oct. 1994. *Lecture Notes in Computer Science*, 866:6-15, Springer-Verlag.

[WK88] D. Whitley and J. Kauth. Genitor: A different genetic algorithm. In *Rocky Mountain Conference on Artificial Intelligence*, pages 118–130, 1988.

Appendix: Multi-Start PLG vs. PLG-GBA

To compensate for the difference in the running time of algorithms, we run PLG with many different initial bisections and return the best. We denote by Multi-Start PLG the algorithm that tries 500 runs of PLG and produces the best as its final solution. Table 3 compares the performance of Multi-Start PLG and PLG-GBA on the random graphs. For the other graphs, both of them found the best known on the average with few exceptions. PLG-GBA outperformed Multi-Start PLG for all random graphs. The results show the effectiveness of the genetic hybrid search process.

Table 3: Comparison of Multi-Start PLG and PLG-GBA

Graph	Multi-Start PLG[1] Ave[2]	Multi-Start PLG[1] CPU	PLG-GBA Ave[2]	PLG-GBA CPU
G500.2.5	51.44	2.91	**50.41**	2.48
G500.05	219.48	6.87	**218.04**	5.60
G500.10	629.35	14.59	**626.85**	12.05
G500.20	1749.09	32.32	**1745.59**	29.42
G1000.2.5	98.55	8.26	**96.23**	6.81
G1000.05	456.41	19.46	**449.49**	22.89
G1000.10	1377.15	40.97	**1364.42**	45.16
G1000.20	3397.49	86.34	**3384.49**	77.90

1. Each run is the best of 500 runs of PLG.
2. 1,000 runs on Alpha PC 600 MHz.