Journal of Circuits, Systems, and Computers, Vol. 2, No. 2 (1992) 159–185
© World Scientific Publishing Company

SIMULATED ANNEALING AND TABU SEARCH ALGORITHMS FOR MULTIWAY GRAPH PARTITION[*][†]

L. TAO, Y. C. ZHAO, K. THULASIRAMAN, and M. N. S. SWAMY
Faculty of Engineering and Computer Science, Concordia University, Montreal, Canada H3G 1M8

Received 21 January 1992
Accepted 31 March 1992

For a given graph G with vertex and edge weights, we partition the vertices into subsets to minimize the total weights for edges crossing the subsets (weighted cut size) under the constraint that the vertex weights are evenly distributed among the subsets. We propose two new effective graph partition algorithms based on simulated annealing and tabu search, and compare their performance with that of the LPK algorithm reported in Ref. 12. Extensive experimental study shows that both of our new algorithms produce significantly better solutions than the LPK algorithm (maximal and minimal improvements on average weighted cut size are roughly 51.8% and 10.5% respectively) with longer running time, and this advantage in solution quality would not change even if we run the LPK algorithm repeatedly with random initial solutions in the same time frame as required by our algorithms.

1. Introduction

Given an undirected graph $G = (V, E)$, an integer m $(1 < m \leq |V|)$, and two weight functions $w_1\colon V \to I$ and $w_2\colon E \to I$ (I is the set of positive integers), an m-way partition π of G is a function $\pi\colon V \to \{1, 2, \ldots, m\}$ such that $V = P_\pi(1) \cup P_\pi(2) \cup \cdots \cup P_\pi(m)$, where $P_\pi(i) = \{v \in V | \pi(v) = i\}$ for $1 \leq i \leq m$. Our objective is to derive m-way partitions π that can minimize

$$W_2(\pi) = \sum_{\substack{e=\{u,v\} \in E \\ \pi(u) \neq \pi(v)}} w_2(e)$$

under the constraint that

$$W_1(\pi) = \sum_{1 \leq i < j \leq m} |w_1(P_\pi(i)) - w_1(P_\pi(j))|$$

is minimal (for any subset $C \subseteq V$, $w_1(C) = \sum_{v \in C} w_1(v)$). We call $w_1(v)$ the *vertex weight*

* This paper was recommended by Regional Editor D. A. Mlynski.

† This work has been partially supported by the Canadian NSERC research grant OGP0041648 and FCAR research grant 92NC0026.

of vertex v, $w_2(e)$ the *edge weight* of edge e, $W_1(\pi)$ the *balance measure*, and $W_2(\pi)$ the *weighted cut size*. Informally, we want to partition the graph vertices into mutually exclusive subsets so that the total weight of the edges crossing the subsets is minimized under the condition that the vertex weights are distributed evenly among the subsets.

Our m-way graph partition is a natural generalization of the classical *graph bisection problem*,[10] in which $|V|$ is even, w_1 is a constant function, and $m = 2$. Like the graph bisection problem, m-way graph partition has important applications in VLSI design[14] and parallel processing.[1,3] If we use the vertices to model processes with w_1 specifying their computation time, and edges the logical interprocess communication channels with w_2 specifying their message traffic load, then an optimal solution to the m-way graph partition problem will allow us to distribute the computations evenly among a set of m identical (or comparable in computation power) processors interconnected by a bus to minimize the interprocessor message traffic on the bus, which can be the bottleneck of the system performance. On the other hand, if we use the vertices to represent the electrical components with w_1 specifying their size, and the edges the interconnections between the components with w_2 specifying the signal traffic load between the components, then an optimal solution to the m-way graph partition problem will allow us to distribute evenly the components on m chips and minimize the interchip signal traffic.

Since the graph partition problem is NP-hard,[5] efficient and effective heuristic algorithms have been studied for its solutions. Most of these attempts are based on one of the following three basic approaches:

- **Kernighan-Lin heuristics.** This approach[10] is characterized by repeated, exhaustive searches for sequences of moves to improve the current solution. It is in general aggressive, fast, but easy to get stuck in local optima. The Kernighan-Lin algorithm for graph bisection,[10] which has been the main competitor for this problem for almost two decades, is a typical example for this approach. For graphs with small edge weight range and small average degree, a special *bucket* data structure can improve the speed of this algorithm.[4] Recently the Kernighan-Lin approach was also applied to the more general m-way graph partition problem,[12] which out-performs the simple Kernighan-Lin m-way graph partition algorithm outlined in Ref. 10.
- **Simulated annealing.** This is a stochastic optimization approach based on the analogy to physical annealing.[2,11] It tries to avoid being trapped in local optima by accepting both "good" and "bad" moves at the beginning of the iterations, and gradually lowering the probability of accepting "bad" moves. Even though in theory simulated annealing can find global optima if we lower the above probability slowly in exponential time,[8] its performance in a practical time frame depends heavily on the parameters comprising its "cooling schedule." Recently Johnson *et al.*[9] made a critical evaluation for the performance of the simulated annealing approach to the graph bisection problem and compared its performance with that of the Kernighan-Lin approach. In general, simulated annealing is time-consuming, but it has been successfully applied to many optimization problems.

- **Tabu search.** This is a new approach to combinatorial optimization characterized by aggressive local search during each iteration, and avoiding cycling in the solution space by keeping a short history of the attributes of the recent solutions.[6,7] A tabu search algorithm for m-way graph partition has been reported with limited success.[13] In general, tabu search algorithms are slower than other problem-specific heuristics, but they have been successfully applied to many problem domains. The relative performance of simulated annealing and tabu search is problem-dependent and inconclusive at this stage.

Based on our experience reported in Ref. 13, we find that the design of solution neighborhood is a critical issue in any iterative approximation solutions to the m-way graph partition problem. Vertex moves or vertex exchanges alone cannot produce satisfactory solution quality. In this paper we design two new graph partition algorithms based on simulated annealing and tabu search using a new definition of solution neighborhood. This new neighborhood structure represents a good trade-off between the aggressiveness and the efficiency of the search process. To demonstrate the power of our new neighborhood design and algorithms, we conduct excessive experimental studies to compare the performance of the new algorithms with the one based on the Kernighan-Lin approach reported in Ref. 12. We use both random graphs and geometric graphs as our benchmark graphs. While most studies of graph partition in the literature use constant functions for w_1 and w_2, we conduct performance comparisons both for graphs with constant vertex and edge weights and for graphs with varying vertex and edge weights. We believe that the latter are more in line with some practical applications. Experiments show that both of our new algorithms produce significantly better solutions than the LPK algorithm with longer running time, and the one based on simulated annealing is more stable and superior to the one based on tabu search. Compared with the average $W_2(\pi)$ for LPK, the average improvements of $W_2(\pi)$ over all of our benchmark geometric graphs are 51.8% for simulated annealing and 42.9% for tabu search respectively; the average improvements of $W_2(\pi)$ over all of our benchmark random graphs are 11.6% for simulated annealing and 10.5% for tabu search respectively. For each of the problem instances, all the algorithms reach the same minimal $W_1(\pi)$. The advantage in solution quality of our algorithms would not change even if we run the LPK algorithm repeatedly with random initial solutions in the same time frame as required by our algorithms.

The paper is organized into six sections. Section 2 addresses design problems specific to the graph partition problem. These problems are independent of particular approaches or algorithms. Our neighborhood design (S_3) in this section is mainly responsible for the good performance of our new algorithms. Section 3 abstracts the Kernighan-Lin heuristics into a general approach and puts the LPK algorithm,[12] our main competitor, into spectrum. In Sec. 4 we adapt the simulated annealing and tabu search approaches to the m-way graph partition problem after we summarize the basic mechanisms underlying them. Intensive and extensive experimental studies are reported in Sec. 5. Section 6 concludes the paper with a few observations.

2. Problem-Specific Design Issues

From an abstract point of view, all the algorithms in this paper perform a series of iterations. During each iteration, a subset of the neighborhood of the current solution in the solution space is investigated and the current solution is updated accordingly (making a *move*). To make these algorithms efficient and effective, following are some common necessary conditions:

1. Efficient evaluation of the objective function $W_2(\pi)$ and the constraint condition for $W_1(\pi)$. Since the problem to partition a graph with uneven vertex weights to minimize $W_1(\pi)$ alone is NP-hard in general, we cannot expect heuristic algorithms to generate feasible solutions all the time. In practice we have to treat $W_1(\pi)$ and $W_2(\pi)$ as two objective functions, and try to minimize both of them while giving $W_1(\pi)$ much higher priority. It will be very helpful if we can combine these two conflicting objectives into a single one based on some trade-offs between them.
2. The moves applied in the algorithms should allow any current solution to reach any other solution, and an appropriate neighborhood of each solution in the solution space should be defined to compromise the neighborhood search time and the aggressiveness of each iteration.
3. A gain function should be defined for all the allowed moves to measure the effectiveness of each potential move, and this gain function should allow incremental update following each move to minimize the computational overhead introduced by re-evaluating the gain function.

These issues are treated separately in the following subsections.

2.1. *Graph transformation*

The time complexity of an iterative algorithm is largely determined by the efficiency by which the objective functions and the constraint conditions are evaluated. While $W_2(\pi)$ allows simple incremental update after each vertex move or vertex exchange operation, $W_1(\pi)$ needs at least $O(m)$ update steps after each of such operations. Therefore we adopt the following graph transformation in Ref. 12 to combine $W_1(\pi)$ and $W_2(\pi)$ into a single objective function easy for incremental evaluation.

Transformation algorithm:
Given a graph $G = (V, E)$ described in the last section, we transform G into another graph $G^* = (V, E^*)$ where $E^* = \{\{u, v\} | u, v \in V\}$, and define a new edge weight function $w_3: E^* \rightarrow \Re$ ($\Re$ is the set of all positive real numbers) such that

$$w_2(e) = \begin{cases} w_1(u)w_1(v)R - w_2(e) & \text{if } e = \{u, v\} \in E; \\ w_1(u)w_1(v)R & \text{if } e = \{u, v\} \in E^* - E \end{cases}$$

where R is a positive real number called the *augmenting factor*.

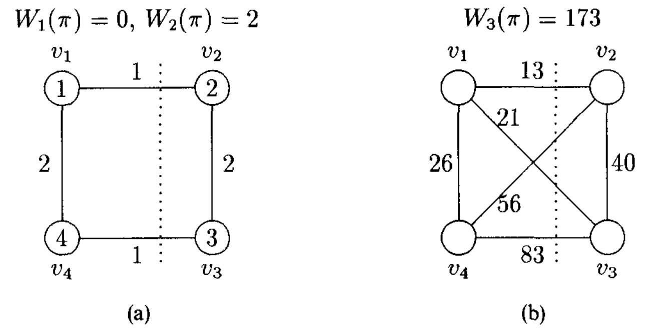


Fig. 1. Example for graph transformation.

As pointed out in Ref. 12, if $R > \sum_{e \in E} w_2(e)$, any partition π that maximizes

$$W_3(\pi) = \sum_{\substack{e=\{u,v\} \in E^* \\ \pi(u) \neq \pi(v)}} w_3(e) = R \sum_{1 \leq i < j \leq m} w_1(P_\pi(i)) w_1(P_\pi(j)) - W_2(\pi)$$

will minimize $W_2(\pi)$ under the constraint that $\sum_{1 \leq i < j \leq m} w_1(P_\pi(i)) w_1(P_\pi(j))$ is maximized, which in turn is equivalent to minimizing $W_1(\pi)$. The latter part of this argument is not covered in Ref. 12, and we prove it in Appendix A to this paper. Figure 1 gives an example for the graph transformation (the vertex weights are marked inside the vertices, the edge weights are marked along the edges, and the two partitions are separated by a dotted line). We use $R = 7$.

Based on the above transformation, from now on, we will focus on graph partitions π that maximize $W_3(\pi)$.

2.2. *Move and neighborhood design*

Let X be the set of all mappings $V \to \{1, 2, \ldots, m\}$. We call X the *solution space*. Our transformed problem can be presented as

$$\text{Maximize} \qquad W_3(\pi): \quad \pi \in X$$

where $W_3(\pi)$ is the objective function.

A wide range of heuristic algorithms for solving problems capable of being written in this form can be characterized conveniently by reference to sequences of *moves* that lead from one trial solution (selected $\pi \in X$) to another. Let S be the set of all defined moves. We use $S(\pi)$ ($\pi \in X$) to denote the subset of moves in S applicable to π. For any $s \in S(\pi)$, $s(\pi)$, the new solution obtained by applying move s to π, is called a *neighbor* of π. If $s(\pi) \neq s'(\pi)$ for any pair of different moves $s, s' \in S(\pi)$, we can use $|S(\pi)|$ to denote the *neighborhood size* of solution π.

To optimize the algorithm performance, S should be defined with the following properties:

- **Reachability:** Given any two solutions π and π' in X, it should be possible to apply a sequence of moves in S to reach π' from π. This property will greatly increase the probability for an algorithm to converge to the global optimum.
- **Efficiency:** Given any solution $\pi \in X$ and $s \in S$, the cost of $s(\pi)$ can be easily evaluated by incrementally updating the cost of π. This will allow us to avoid evaluating the cost function $W_3(\pi)$ during each iteration, an operation having time complexity $O(n^2)$.
- **Injectiveness:** Given two different moves $s, s' \in S$, for any $\pi \in X$, $s(\pi) \neq s'(\pi)$. This will make sure that each neighbor of the current solution will be checked only once for the current neighborhood search.

Vertex move and *vertex exchange* are two popular classes of moves for graph partition. Let S_1 be the set of all moves for moving one vertex away from its current partition, and S_2 the set of all moves for exchanging two vertices belonging to two different partitions. Both S_1 and S_2 enjoy the injectiveness property. The cost of the current solution can be incrementally updated for moves from both S_1 and S_2, as will be explained in the next subsection. Many graph partition algorithms in the literature (Refs. 12, 9, 4, for example) favor S_1 because it has a smaller neighborhood size $n(m - 1)$, while the average neighborhood size for S_2 is $O(n^2)$. We can make the following two further observations about the reachability of S_1 and S_2.

1. S_1 also enjoys the reachability property. But if we only allow vertex moves that will not worsen the cost of the current partition by $R \cdot \max(w_1) - (\max(w_2) - \min(w_2))$ or more (this is the case when the simulated annealing is in its low-temperature phases, or when the tabu search always has moves with gains of smaller absolute value), then this reachability cannot always be realized. For example, for the graph bisection of G_1 in Fig. 2(a), no sequence of vertex moves can transform it to the optimal bisection of G_1 in Fig. 2(c) unless we accept moves that will reduce $W_3(\pi)$ by 6. Figure 2 (b) shows an example vertex move for the bisection in Fig. 2(a).
2. In general S_2 does not have the reachability property. For example, given the graph bisection of G_2 in Fig. 3(a), vertex exchanges will never lead us to the optimal

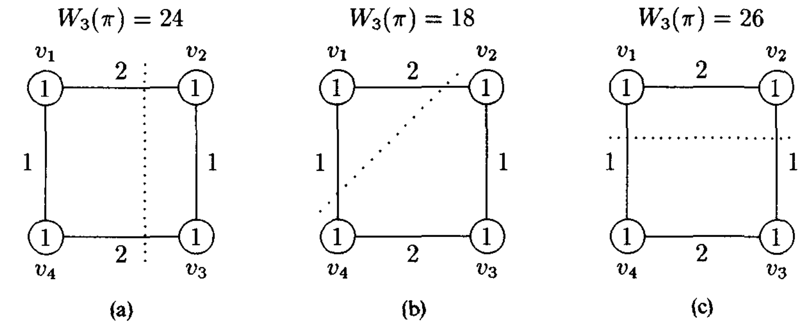


Fig. 2. Example partitions of G_1, $R = 7$.

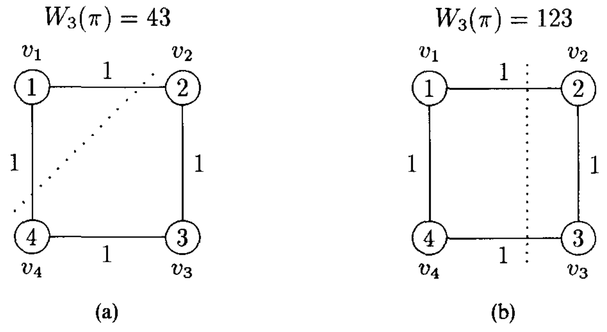


Fig. 3. Example partitions of G_2, $R = 5$.

bisection of G_2 in Fig. 3(b) (while vertex moves do) because they cannot change the cardinality of each partition. However, we can easily transform the bisection in Fig. 2(a) to that in Fig. 2(c) by exchanging vertices v_2 and v_4.

Our experiments show that vertex moves in S_1 are very effective in balancing the vertex weights among the partitions to minimize $W_1(\pi)$, while vertex exchanges in S_2 are very effective in refining the partition to minimize $W_2(\pi)$. The best order and mixture of vertex moves and vertex exchanges are problem instance dependent. To compromise the neighborhood size and the effectiveness of the moves, our algorithms use a special set S_3 of moves, where for any $\pi \in X$, $S_3(\pi) = S_1(\pi) \cup S_2'(\pi)$, and

$$S_2' = \{\text{exchange } u \text{ and } v \mid v \in V, \text{ moving } v \text{ to } P_\pi(j) \text{ maximizes gain which is } < 0;\ u \in P_\pi(j)\}.$$

Informally, we give vertex moves higher priority than vertex exchanges. For a given partition π and a given vertex $v \in P_\pi(i)$, we first try moving v to all the other partitions; if moving v to $P_\pi(j)$ has the best gain which is less than zero, then we also try exchanges of v with each of the vertices in $P_\pi(j)$. Experiments show that S_3 performs better than S_1 or S_2 alone in terms of both running time and solution quality for all of our new graph partition algorithms.

2.3. *Incremental gain update*

Given the current partition π and a move $s \in S(\pi)$, we call $W_3(s(\pi)) - W_3(\pi)$ the *gain* of move s. Many algorithms need to choose moves with maximum gains during each iteration. It will be very helpful if we have a *gain function* that can easily be incrementally updated after each move.

Given a partition π of G, for any $s \in S_1$, the gain for moving any vertex $v \in P_\pi(i)$ to partition $P_\pi(j)$ $(1 \le i, j \le m)$ can be defined to be

$$g_1(v,j) = \sum_{u \in P_\pi(i)} w_3(\{u,v\}) - \sum_{u \in P_\pi(j)} w_3(\{u,v\});$$

and for any $s \in S_2$, the gain for exchanging any pair of vertices $u \in P_\pi(i)$ and $v \in P_\pi(j)$ $(i \neq j)$ can be defined to be

$$g_2(u, v) = g_1(u,j) + g_1(v, i) + 2w_3(\{u, v\})$$

because we can view the vertex exchange as consisting of two consecutive vertex moves. Since g_2 can be defined in terms of g_1, in the following we only need to consider the incremental update of g_1 after each vertex move.

Based on the definition of g_1, it can be verified that after moving vertex $v \in P_\pi(i)$ to partition $P_\pi(j)$ $(i \neq j)$, the gain function g_1 can be incrementally updated as follows:

$$\begin{aligned}
g_1'(v, i) &= -g(v,j) & & \\
g_1'(v, k) &= g(v, k) - g(v,j), & & k \notin \{i,j\} \\
g_1'(u,j) &= g(u,j) - 2w_3(\{u, v\}), & & \forall u \in P_\pi(i) \\
g_1'(u, k) &= g(u, k) - w_3(\{u, v\}), & & \forall u \in P_\pi(i),\ k \notin \{i,j\} \\
g_1'(u, i) &= g(u, i) + 2w_3(\{u, v\}), & & \forall u \in P_\pi(j) \\
g_1'(u, k) &= g(u, k) + w_3(\{u, v\}), & & \forall u \in P_\pi(j),\ k \notin \{i,j\} \\
g_1'(u, i) &= g(u, i) + w_3(\{u, v\}), & & \forall u \notin P_\pi(i) \cup P_\pi(j) \\
g_1'(u,j) &= g(u,j) - w_3(\{u, v\}), & & \forall u \notin P_\pi(i) \cup P_\pi(j)
\end{aligned}$$

where g_1' marks the new value of g_1.

3. LPK Algorithm Based on Kernighan-Lin Approach

In this section we introduce the LPK graph partition algorithm proposed by Lee, Park, and Kim.[12] We present it as a special adaptation of the more general Kernighan-Lin approach, which in turn is an improvement of the basic *local search* algorithm outlined in Fig. 4.

While simple and only able to find local optima, the local search is the backbone of all the other algorithms studied in this paper. These algorithms all try to improve the local search by accepting conditionally the worse moves, a necessary condition to find global optima. But they differ at when and how to introduce these "downhill" moves. In this section we first introduce the main principles of the Kernighan-Lin approach, from which the LPK algorithm is derived.

```
1. Get an initial solution $\pi$.
2. While there is an untested neighbor of $\pi$ do:
   2.1. Let $\pi'$ be an untested neighbor of $\pi$.
   2.2. If $W_3(\pi') > W_3(\pi)$, set $\pi = \pi'$.
3. Return $\pi$.
```

Fig. 4. Local search.

```
1. Get a random initial solution $\pi$.
2. Let $k = 1, \tilde{g}_0 = 0$.
3. Repeat
   3.1. Let $\tilde{\pi}_0 = \pi, T = \emptyset, i = 1$.
   3.2. While $S(\tilde{\pi}_{i-1}, V - T) \neq \emptyset$ do:
        3.2.1. Let $s_i$ be the move in $S(\tilde{\pi}_{i-1}, V - T)$
               maximizing $\Delta$ in 3.2.2.
        3.2.2. Let $\tilde{\pi}_i = s_i(\tilde{\pi}_{i-1}), \Delta = W_3(\tilde{\pi}_i) - W_3(\tilde{\pi}_{i-1})$.
        3.2.3. Let $\tilde{g}_i = \tilde{g}_{i-1} + \Delta, T = T \cup V(s_i)$.
   3.3. Let $\tilde{g}_k = \max\{\tilde{g}_1, \tilde{g}_2, \ldots, \tilde{g}_{i-1}\}$.
   3.4. Let $\pi = \tilde{\pi}_k$.
   Until $\tilde{g}_k \leq 0$.
4. Return $\pi$.
```

Fig. 5. Kernighan-Lin heuristic.

3.1. *Kernighan-Lin approach*

The Kernighan-Lin approach differs from the local search at two key aspects: (1) It is more aggressive during each iteration: it always uses the move among the current candidates that can maximize the gain; (2) It gives each vertex a chance to move, and the "downhill" moves will be accepted as long as the compound gain of the sequence of moves including these "downhill" ones is positive.

Let $V(s)$ be the subset of vertices in V involved in the move s, and $S(\pi, C)$ ($\pi \in X$, $C \subseteq V$) the subset of moves in $S(\pi)$ that only redefines $\pi(v)$ for $v \in C$. We can present the generic Kernighan-Lin algorithm in Fig. 5. Given a random initial solution (π), the algorithm executes a main loop (Step 3) which will stop only if no positive compound gain can be found in its last iteration. During each iteration of this main loop, an inner loop (Step 3.2) is used to move each vertex exactly once, and T is used to maintain the vertices already involved in some previous moves in this main iteration. The ith iteration of the inner loop makes one move (s_i), updates the current solution ($\tilde{\pi}_i$), and calculates the compound gain up to that stage ($\tilde{g}_i$). The chosen move (s_i) must maximize the gain (Δ) and involve no vertices already moved in the current main iteration ($s_i \in S(\tilde{\pi}_{i-1}, V - T)$). After this inner loop the solution corresponding to the maximum compound gain ($\tilde{g}_k$) is used as the initial solution for the next iteration of the main loop.

3.2. *LPK algorithm*

While the framework in Fig. 5 has been applied to different problems before,[10] it is Lee, Park and Kim[12] who first applied it to the m-way graph partition problem in conjunction with their graph transformation technique. They used S_1 to define moves and neighborhood structure, leading to a time complexity of each main iteration to be $O(mn^2)$. Even though the LPK algorithm is designed for multiway graph partition, their experiments focused on graph bisections for graphs with constant vertex and edge weights.

4. Simulated Annealing and Tabu Search Algorithms

Simulated annealing and tabu search are two of the most important techniques for general combinatorial optimization. Even though they are new (having a history less than 10 years) and still under development, they have claimed success in many application domains. In this section we summarize the main ideas of these two approaches, and present our adaptations of them to the m-way graph partition problem. Our experiments show that the graph partition algorithms based on these two approaches are very competitive in terms of solution quality (see Sec. 5). Our unique solution neighborhood structure defined by S_3 is partially responsible for their success.

4.1. *Simulated annealing*

Our first attempt to design a new graph partition algorithm to outperform the LPK algorithm is based on the simulated annealing approach. Simulated annealing can be viewed as an enhanced version of the local search. It attempts to avoid entrapment in poor local optima by allowing occasional downhill moves. This is done under the influence of a random number generator and a control parameter called the *temperature*. As typically implemented,[9] the simulated annealing approach involves a pair of nested loops and two additional parameters, a *cooling ratio* r, $0 < r < 1$; and an integer *temperature length* L (see the generic simulated annealing algorithm in Fig. 6).

The heart of this procedure is the loop at Step 3.1. Note that $e^{\Delta/T}$ will be a number in the interval $(0, 1)$ when $T > 0$ and $\Delta < 0$, and rightfully can be interpreted as a probability that depends on Δ and T. The probability that a downhill move will be accepted diminishes as the temperature declines, and, for a fixed temperature T,

```
1. Get a random initial solution $\pi$.
2. Get an initial temperature $T > 0$.
3. While stop criterion not met do:
   3.1. Perform the following loop $L$ times:
        3.1.1. Let $\pi'$ be a random neighbor of $\pi$.
        3.1.2. Let $\Delta = W_3(\pi') - W_3(\pi)$.
        3.1.3. If $\Delta \geq 0$ (uphill move),
               set $\pi = \pi'$.
        3.1.4. If $\Delta < 0$ (downhill move),
               set $\pi = \pi'$ with probability $e^{\Delta/T}$.
   3.2. Set $T = rT$ (reduce temperature).
4. Return the best $\pi$ visited.
```

Fig. 6. Simulated annealing.

small downhill moves have higher probabilities of acceptance than larger ones. This particular method of operation is motivated by a physical analogy, best described in terms of the physics of crystal growth.[11] It has been proven that the algorithm will converge to a global optimum if the temperature is lowered exponentially and the initial temperature chosen is sufficiently high.[8]

There are two main issues related to the adaptation of this general approach to the m-way graph partition problem. The first is the design of moves and neighborhood structure, the other is the design of the cooling schedule. We use S_3 (see Subsec. 2.2) as the set of moves. More specifically, during each iteration, we randomly choose two partitions $P_\pi(i)$ and $P_\pi(j)$ $(i \neq j)$, then randomly choose a vertex $v \in P_\pi(i)$. If moving v to $P_\pi(j)$ has a non-negative gain, then we use its resulting mapping as π'; otherwise we randomly choose a vertex $u \in P_\pi(j)$ and try to exchange u and v, and use the mapping resulting from the move with better gain as π'.

As for the cooling schedule design, we made the following decisions.

1. We let $L = n \cdot$ SIZEFACTOR, where SIZEFACTOR is a parameter.
2. The initial temperature T_0 is chosen so that the initial acceptance rate is around *INITPROB*, another parameter in the range (0, 1).
3. For each temperature we measure the acceptance rate of the proposed moves. The algorithm stops when for five temperatures the acceptance rate is lower than *MINPERCENT* and the best visited solution is not improved in that period of time. Here MINPERCENT is another parameter in the range (0, 1).

All the parameters for our simulated annealing algorithm are not independent. We tune the parameters of our annealing algorithm for each of our benchmark graphs one at a time. We repeat the process until no perturbation of the parameters can improve the performance. As an example, for our benchmark graph G600 (specified in Table 3), we find that one set of satisfactory parameter values are $r = 0.95$, SIZEFACTOR = 16, INIPROB = 0.4, and MINPERCENT = 0.02.

4.2. *Tabu search*

Tabu search is another newer general approach for combinatorial optimization.[6,7] It differs from simulated annealing at two main aspects:

- It is more aggressive. For each iteration the whole neighborhood of the current solution is usually searched exhaustively to find the best candidate moves.
- It is deterministic. Each iteration repeats the above exhaustive search for best candidate moves. The best candidate move which does not cause cycling in the solution space will be used no matter what sign its gain has. A *tabu list* is usually used to record the recent move history to avoid solution cycling, hence the name of the approach.

Figure 7 outlines a generic tabu search algorithm using π to represent a solution, W_3 the cost function, and t the length of the tabu list. Given a random solution, the algorithm repeats the loop at Step 2 until some stop criterion is met. During each iteration, the algorithm makes an exhaustive search of the solutions in the neighborhood of the current solution which have not been traversed in the last t ($t > 1$) iterations. The neighboring solution with the best cost will be used to replace the current solution. The main design issues for a tabu search algorithm are as follows:

1. The design of the neighborhood (moves) of the current solutions. A large neighborhood usually makes each iteration more aggressive but also more time-consuming.
2. The design of the contents of the tabu list. If move s is used to transform the current solution to π, the corresponding cell of the tabu list should capture some attributes of π or s so that π will not be traversed again in the next t steps. At one extreme, we can store solution π directly in the tabu list. But in practice, to save memory space and checking time, some attributes of s will be stored in the tabu list to prevent s or s^{-1} to be used in the next t iterations. If we use a more detailed set of attributes of a solution or move in each cell of the tabu list, more memory space and checking time will be incurred during the solution-space search, and the searches will be less restrictive since less solutions (in addition to the ones visited in the last t iterations) will be tabued. On the other hand, if we use a more abstract (simplified) set of attributes of a solution or move in each cell of the tabu list, the implementation will

```
1. Get a random initial solution $\pi$.
2. While stop criterion not met do:
    2.1. Let $\pi'$ be a neighbor of $\pi$ maximizing
         $\Delta = W_3(\pi') - W_3(\pi)$ and not visited
         in the last $t$ iterations.
    2.2. Set $\pi = \pi'$.
3. Return the best $\pi$ visited.
```

Fig. 7. Tabu search.

be more space and time efficient for each iteration, and the searches will be more restrictive since more extra solutions will be tabued.

3. The design of the aspiration level function. To make the implementation more space and time efficient, most designs of the contents of the tabu list will tabu too many solutions in addition to those visited in the last t iterations, thus risk to lose good move candidates. As a make-up, we can define an aspiration level $A(s, \pi)$ (usually an integer) for each pair of move s and solution π such that if $W_3(s(\pi)) > A(s, \pi)$ the tabu status of s for the current solution π can be overriden. In practice some attributes of π, instead of π itself, will be used in the definition of $A(s, \pi)$. $A(s, \pi)$ is designed to capture the common properties of the earlier applications of s to solutions sharing the same attribute values as π.
4. The design of the length t of the tabu list. Parameter t determines how long the move history will be saved in the tabu list. Suppose that π is a local optimum, and it needs at least t' consecutive "downhill" moves to go to another local optimum π'. Then $t \geq t'$ is a necessary condition for π to reach π'. In general, the longer the tabu list, the more time for tabu status checking for each move, and the more restrictive the search process. On the other hand, a too short tabu list risks to introduce cycling in the solution space. Parameter t can be a constant or a variable during the execution of the algorithm. For many applications, a tabu list length around 7 is found appropriate.[6]

The following is a description of our tabu search algorithm for m-way graph partition.

1. We use S_3 of Subsec. 2.2 to define the moves and the neighborhood of the current partition.
2. For the tabu list design, we use a circular list to maintain the vertices moved (exchanged) in the last t ($t > 1$) iterations. We find that a more detailed characterization of the past moves usually traps the search process in a small subspace of the solution space (many vertices may never be moved). A constant tabu list length of 5 produces the best performance for most of our problem instances.
3. We use the cost of the best visited solution as the aspiration level $A(s, \pi)$ for all pairs of s and π. Based on the same observation pointed out in the last item, more "flexible" searches implemented by a more sophisticated aspiration level definition tend to limit the real search freedom in the solution space.

In Ref. 13 we reported an old version of the tabu search algorithm for m-way graph partition. Its main difference from the current one is the solution neighborhood design. In that algorithm we first use S_1 to define the solution neighborhood until vertex moves cannot improve the best solution for a limited number of iterations. Then we switch to use S_2 to define the solution neighborhood. When vertex exchanges cannot improve the best solution for a limited number of iterations, the algorithm stops. Such a rigid separation of vertex moves and vertex exchanges greatly limits the freedom of solution movement. The large neighborhood size of S_2 also makes each vertex exchange very time-consuming. In the following Table 1 we compare our new tabu search algorithm (TS) with the old one (TS') using our benchmark graphs R200_4 and R600 (for the

Table 1. Performance comparisons for two tabu search algorithms.

measurements	R200_4		R600	
	TS'	TS	TS'	TS
total move #	127	12	538	336
total exchange #	32	95	164	363
$W_1(\pi)$	0	0	100	100
$W_2(\pi)$	460	358	38309	38122
CPU time (sec.)	551	6	7187	255

Table 2. Characteristics of the random benchmark graphs.

name	n	d	w_1	w_2	$d_{\min}$	$d_{\max}$	$\lvert E\rvert$
R200_4	200	4	1-5	1-5	0	11	410
R200_20	200	20	1-5	1-5	8	34	2026
R400_8	400	8	1-5	1-5	2	18	1625
R400_40	400	40	1-5	1-5	24	57	7927
R600	600	60	1-5	1-5	37	85	17867
R800_16	800	16	1-5	1-5	5	28	6470
R800_80	800	80	1-5	1-5	52	110	31925
R1000_20	1000	20	1-5	1-5	7	34	9979
R1000_100	1000	100	1-5	1-5	72	135	50025

definition of these two graphs, see Table 2). It can be seen that our new tabu search algorithm outperforms the old one with substantially less running time.

5. Experimental Studies

In this section we conduct a thorough experimental study for our two new graph partition algorithms in comparison with the LPK algorithm. The performance comparisons are classified into two categories: (1) intensive study by running each algorithm 1000 times for two benchmark graphs and reporting the statistical performance data; and (2) extensive study by running each algorithm 10 times for a set of benchmark graphs with varied size, density, and partition number m and reporting the statistical performance data. All computations are performed on a SUN Sparc 2 workstation. To simplify presentation, we use LPK, SA, and TS to denote the LPK algorithm, the Simulated Annealing algorithm, and the Tabu Search algorithm respectively.

5.1. *Benchmark graphs*

We use two general classes of graphs for our performance comparisons: *random graphs* and *geometric graphs*. Both of the two classes of graphs are mainly characterized by two parameters: n, the vertex number; and d, the expected degree for each vertex.

Random graph generation:

Given n and d, define $p = d/(n-1)$. Value p specifies the probability that any given pair of vertices constitutes an edge. The vertex and edge weights are generated randomly in some specific integer ranges.

Geometric graph generation:

Given n and d, define $k = \sqrt{d/(n\pi)}$. The coordinates of n vertices are first generated randomly on a unit square plane. Two vertices share a connecting edge iff the Euclidean distance between them is k or less. The vertex weights are again generated randomly in a specific integer range. The weight for any edge is the ceiling integer of the product of a scale-factor $\mathcal{S}$ and the ratio of the distance between the vertices incident to the edge over k. Figure 8 shows a geometric graph (G600) generated with $n = 600$ and $d = 10$.

All of our benchmark graphs are specified in Tables 2 and 3. The first letter of a graph name designates the graph class: R for random graph, and G for geometric graph. For each graph we specify its vertex number n, expected degree d, range for w_1, range for w_2 (for random graphs), $\mathcal{S}$ (for geometric graphs), minimum degree $d_{\min}$, maximum degree $d_{\max}$, and total edge number $|E|$. The last three entries are measured

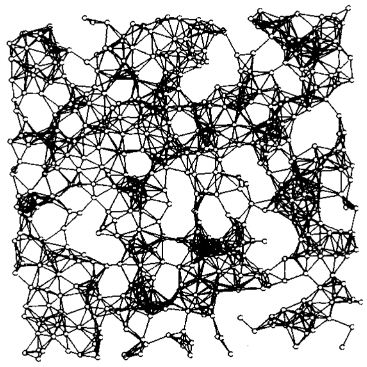


Fig. 8. Geometric graph G600.

Table 3. Characteristics of the geometric benchmark graphs.

name	n	d	w_1	$\mathcal{S}$	$d_{\min}$	$d_{\max}$	$\|E\|$
G200_4	200	4	1–5	10	0	8	387
G200_20	200	20	1–5	10	6	25	1671
G400_8	400	8	1–5	10	1	16	1471
G400_40	400	40	1–5	10	12	53	7006
G600	600	10	1–5	10	1	18	2692
G800_16	800	16	1–5	10	4	31	5807
G800_80	800	80	1–5	10	19	103	27066
G1000_20	1000	20	1–5	10	5	33	9219
G1000_100	1000	100	1–5	10	23	121	42428

from the generated graph. We in general choose small d as most interesting applications involve graphs with a low average degree, and because such graphs are better for distinguishing the performance of different heuristics than denser ones.[9] Although neither of these two classes is likely to arise in a typical application, they provide the basis for repeatable experiments, and, it is hoped, constitute a broad enough spectrum to yield insights into the general performance of the algorithms.

5.2. *Intensive studies*

The performance comparisons for LPK, SA, and TS are complicated by the fact that all of them use random initial solutions. SA also calls the random number generator during its executions. Therefore, for the same graph and the same algorithm, different runs will usually generate different solutions.

To make a fair comparison, we run each of the three algorithms 1000 times with random initial solutions for random graph R600 and geometric graph G600. We set $m = 20$. Table 4 and Table 5 report the $W_1(\pi)$, the average $W_2(\pi)$, the best $W_2(\pi)$, the worst $W_2(\pi)$, the standard deviation for $W_2(\pi)$, and the average running time for each set of these excessive runs. The data for $W_2(\pi)$ is also visualized by the histograms shown in Fig. 9 and Fig. 10. Based on these data we can see that the worst $W_2(\pi)$ for our SA and TS is substantially better than the best $W_2(\pi)$ for LPK; the running time for SA and TS is longer than that for LPK; all algorithms produce the same $W_1(\pi)$ for all runs, a good sign that $W_1(\pi)$ is probably minimized, as required by our model.

Since the running time for LPK is much less than that for SA or TS (roughly 1/8), the $W_2(\pi)$ for one random run of SA or TS should be compared with the expected best $W_2(\pi)$ for multiple (say 8) random runs of LPK. To be more general, we want to find the expected best $W_2(\pi)$ for k random runs of each algorithm for $1 \leq k \leq 100$. Fortunately, instead of repeatedly performing sets of k runs and computing the average of the best, we can derive such data from the existing data for the 1000 runs of each

Table 4. Statistics for 1000 runs of R600.

algorithm	$W_1(\pi)$	ave. $W_2(\pi)$	best $W_2(\pi)$	worst $W_2(\pi)$	std. dev. $W_2(\pi)$	ave. time (sec.)
LPK	100	40373.81	39717	41075	225.50	22.87
SA	100	38265.11	38058	38491	71.61	158.83
TS	100	38321.22	38074	38590	78.56	164.20

Table 5. Statistics for 1000 runs of G600.

algorithm	$W_1(\pi)$	ave. $W_2(\pi)$	best $W_2(\pi)$	worst $W_2(\pi)$	std. dev. $W_2(\pi)$	ave. time (sec.)
LPK	100	1961.17	1732	2139	66.95	18.47
SA	100	541.29	454	640	31.89	189.79
TS	100	688.77	551	833	44.57	184.16

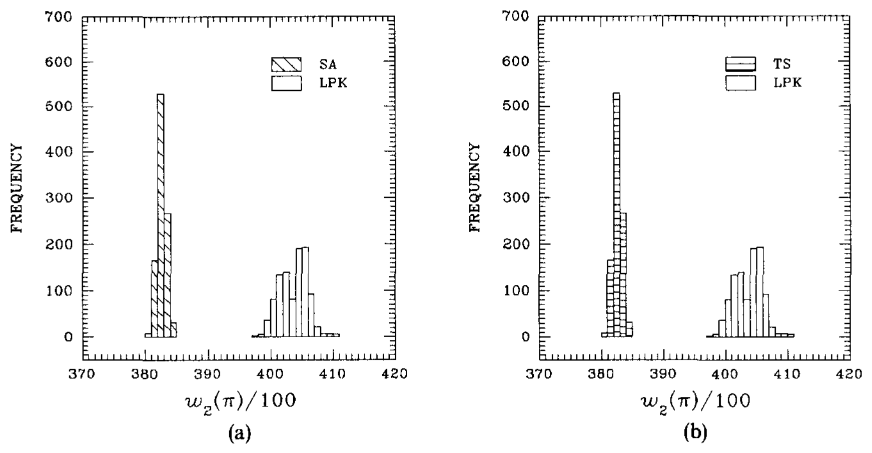


Fig. 9. Histograms for R600.

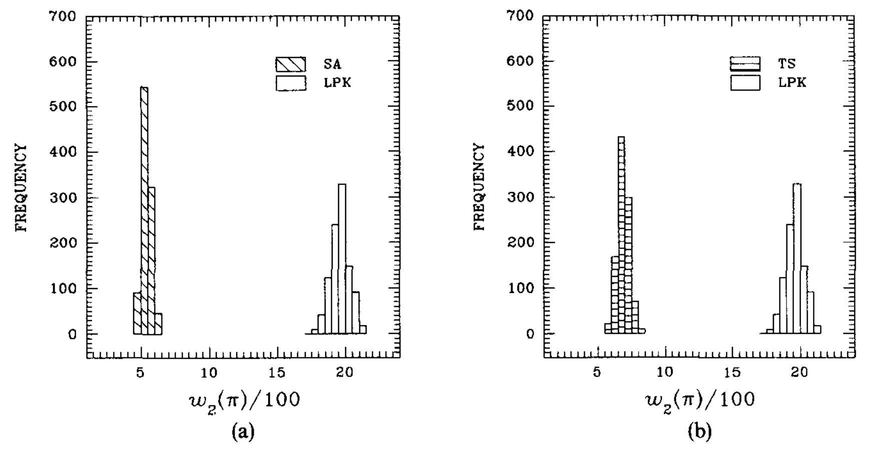


Fig. 10. Histograms for G600.

of the algorithms, as described in Appendix B. The results are summarized in Tables 6 and 7 for R600 and G600 respectively. We can conclude from these two tables that in the same time frame as for running SA or TS once, repeated running of LPK cannot provide better performance.

5.3. *Extensive studies*

To put our results in perspective, we also perform similar, though less intensive, experiments with random graphs and geometric graphs with n ranging from 200 to 1000, d ranging from 4 to 100, and m ranging from 4 to 100. We run each algorithm

Table 6. Expected best $W_2(\pi)$ for k runs of R600.

k	LPK	SA	TS
1	40373.31	38285.11	38321.22
2	40245.66	38224.81	38277.06
5	40107.26	38182.23	38231.26
10	40031.07	38155.18	38202.24
25	39952.72	38124.56	38168.06
50	39900.95	38105.25	38143.91
100	39851.35	38088.99	38120.84

Table 7. Expected best $W_2(\pi)$ for k runs of G600.

k	LPK	SA	TS
1	1961.17	541.29	688.77
2	1923.61	523.34	663.76
5	1882.86	505.49	637.48
10	1855.65	494.68	621.01
25	1823.77	482.64	602.91
50	1802.57	474.90	590.95
100	1783.39	468.37	579.69

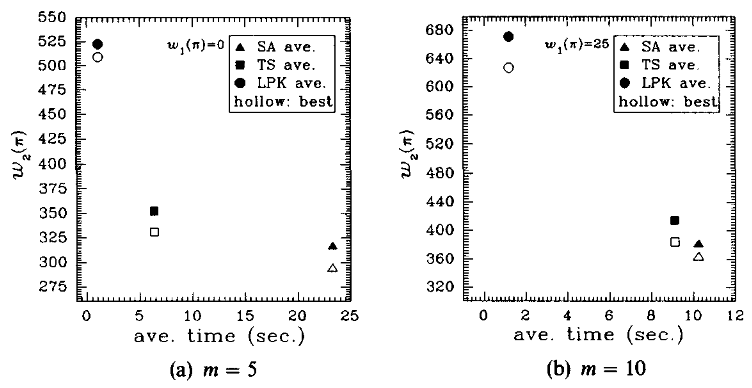
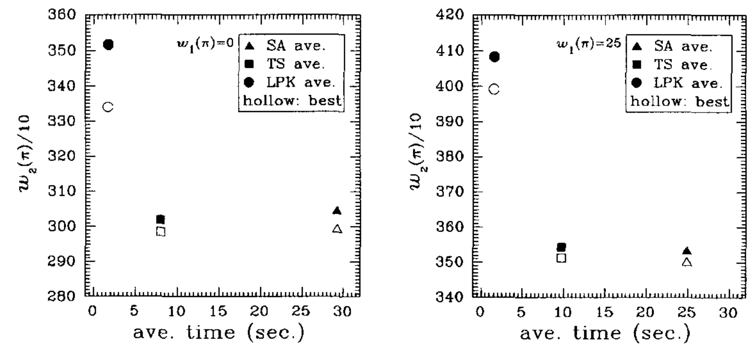


Fig. 11. Performance comparisons for R200_4.

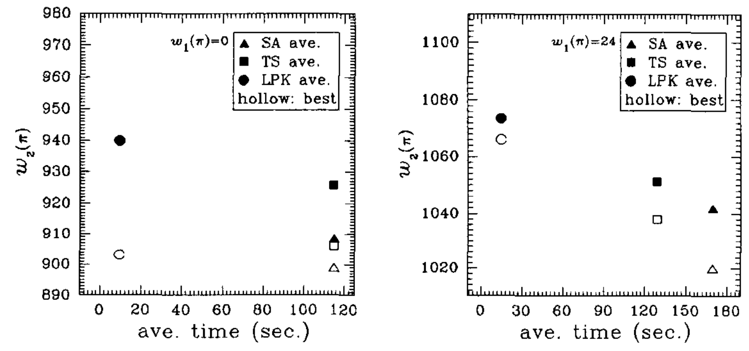
10 times for each graph and report the average $W_2(\pi)$, the best $W_2(\pi)$, and the average running time in Figs. 11 to 18 for random graphs, and Figs. 19 to 26 for geometric graphs. In each figure we use a distinct symbol shape to represent each algorithm, use solid symbols to represent the information on average $W_2(\pi)$, and use hollow symbols to represent the information on the corresponding best $W_2(\pi)$.

We can conclude from these data that for all these problem instances, SA and TS always outperform LPK in terms of solution quality, especially for geometric graphs.



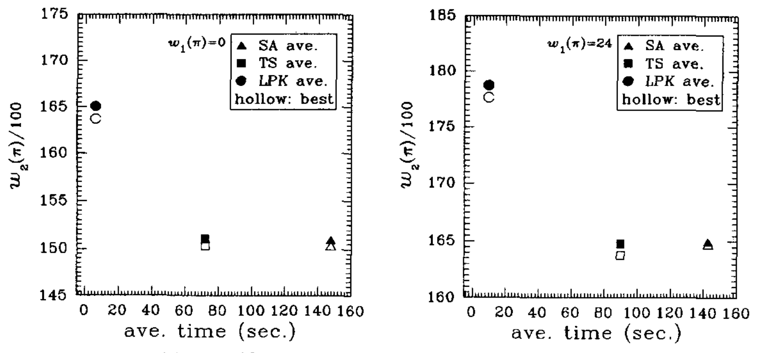
(a) $m = 5$ (b) $m = 10$

Fig. 12. Performance comparisons for R200_20.



(a) $m = 10$ (b) $m = 20$

Fig. 13. Performance comparisons for R400_8.



(a) $m = 10$ (b) $m = 20$

Fig. 14. Performance comparisons for R400_40.

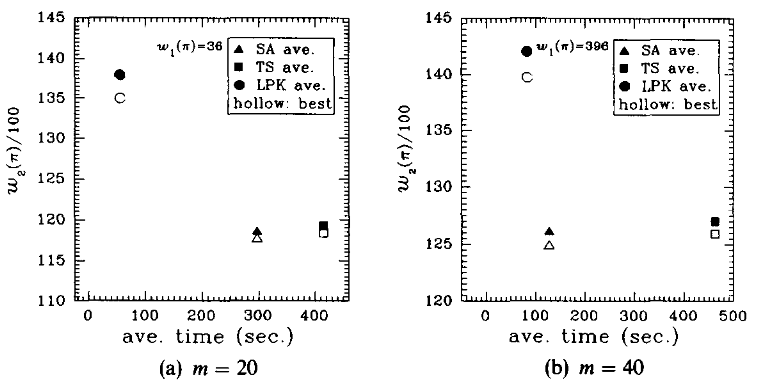


Fig. 15. Performance comparisons for R800_16.

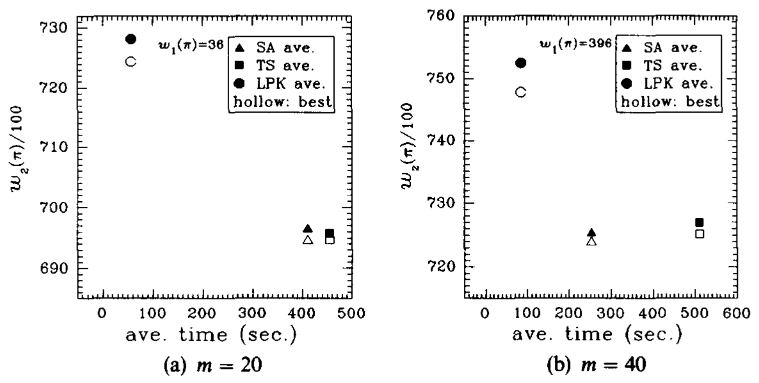


Fig. 16. Performance comparisons for R800_80.

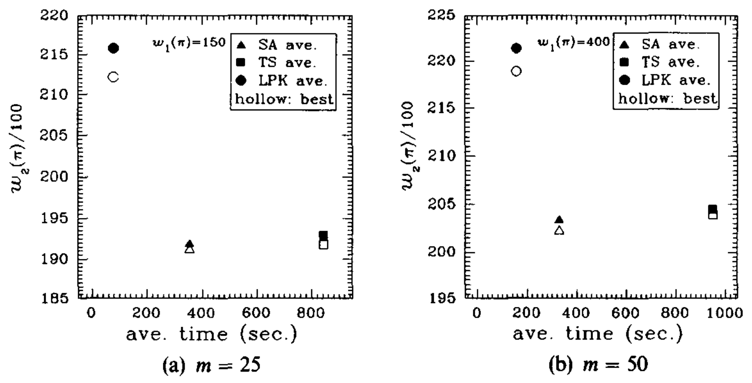
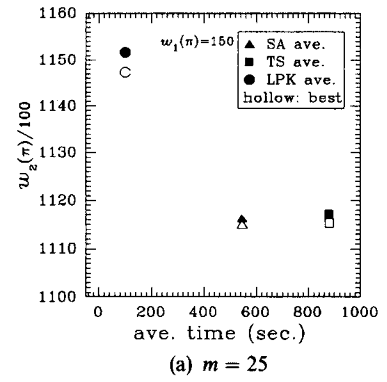
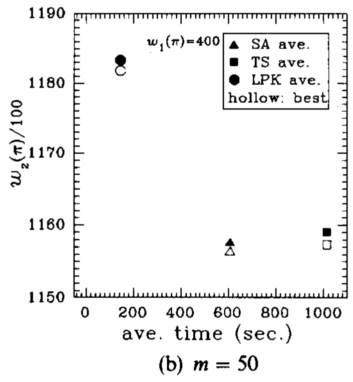


Fig. 17. Performance comparisons for R1000_20.

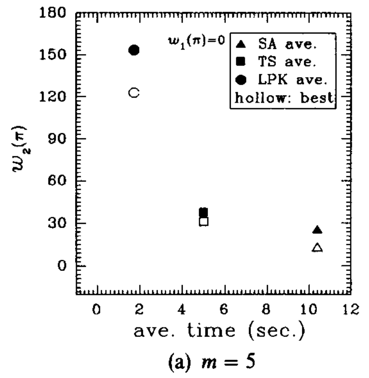


(a) $m = 25$

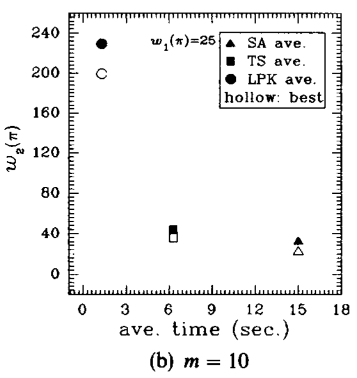


(b) $m = 50$

Fig. 18. Performance comparisons for R1000_100.

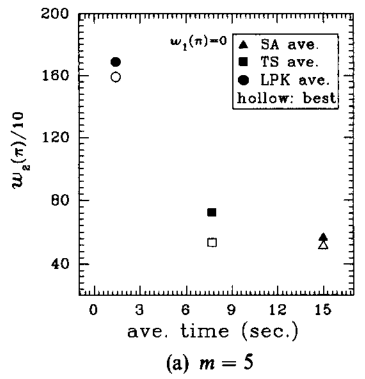


(a) $m = 5$

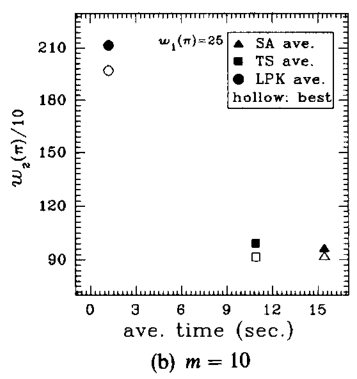


(b) $m = 10$

Fig. 19. Performance comparisons for G200_4.

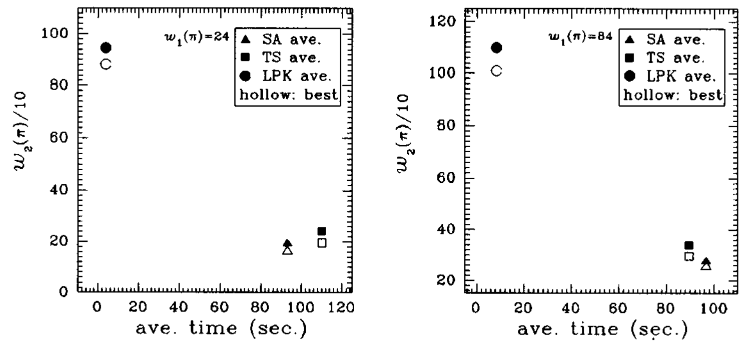


(a) $m = 5$



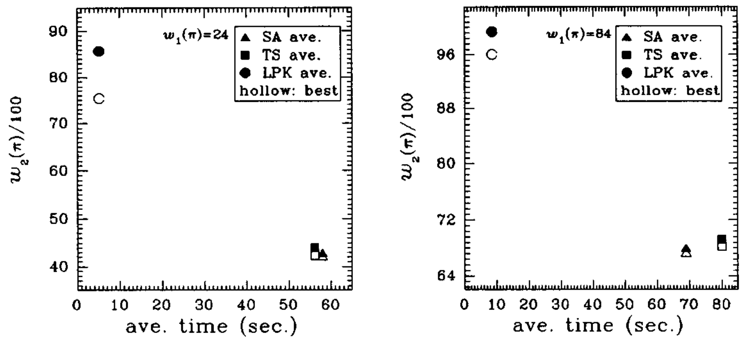
(b) $m = 10$

Fig. 20. Performance comparisons for G200_20.



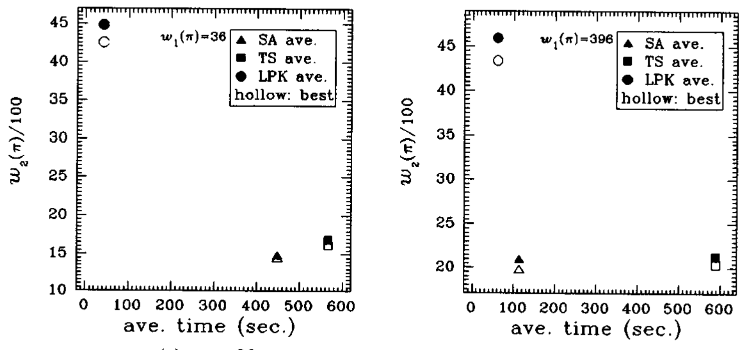
(a) $m = 10$ (b) $m = 20$

Fig. 21. Performance comparisons for G400_8.



(a) $m = 10$ (b) $m = 20$

Fig. 22. Performance comparisons for G400_40.



(a) $m = 20$ (b) $m = 40$

Fig. 23. Performance comparisons for G800_16.

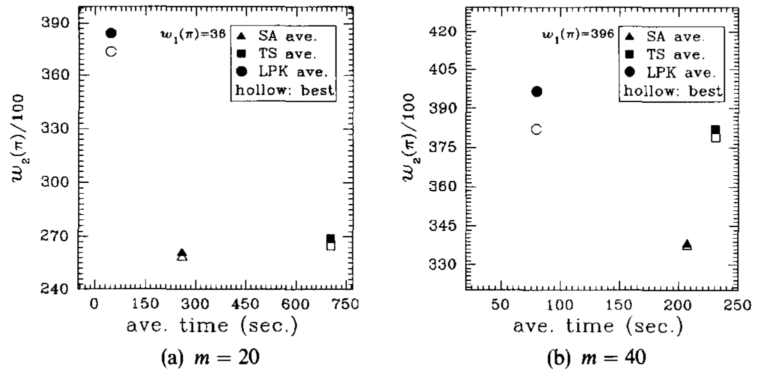


Fig. 24. Performance comparisons for G800_80.

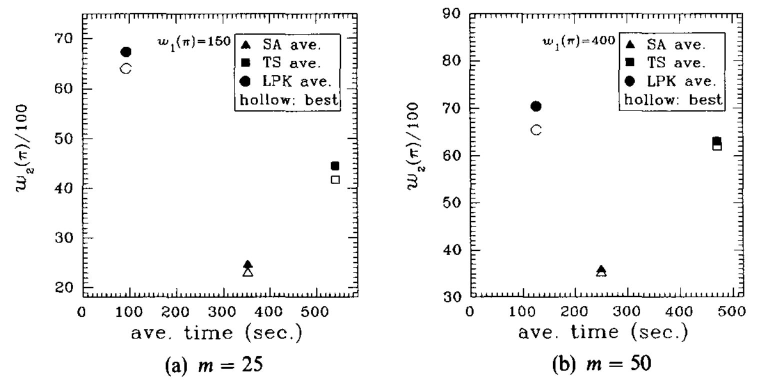


Fig. 25. Performance comparisons for G1000_20.

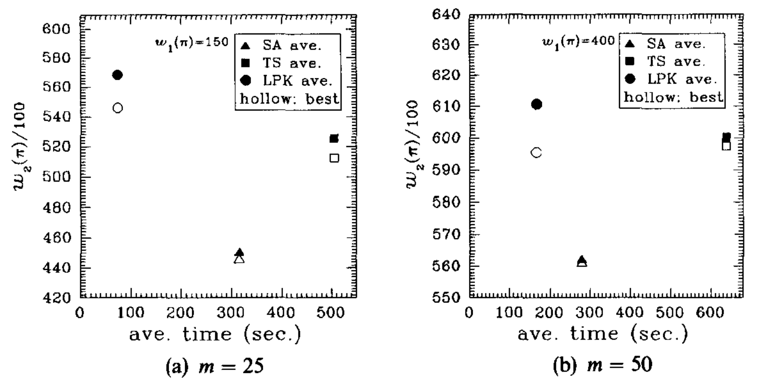


Fig. 26. Performance comparisons for G1000_100.

Compared with the average $W_2(\pi)$ for LPK, the average improvements of $W_2(\pi)$ over all of our benchmark geometric graphs are 51.8% for SA and 42.9% for TS respectively; the average improvements of $W_2(\pi)$ over all of our benchmark random graphs are 11.6% for SA and 10.5% for TS respectively. For each of these problem instances, all the algorithms reach the same minimal $W_1(\pi)$.

6. Conclusion

This study demonstrates that simulated annealing and tabu search are intrinsically more powerful than the Kernighan-Lin approach for the multiway graph partition problem (maximum and minimum average improvements on average weighted cut size are 51.8% and 10.5% respectively). It also shows the importance of the design of the solution neighborhood structure. Based on our experience reported in this paper, we believe that the running time of our algorithms can be greatly reduced if we combine the aggressive search in the tabu search approach with the stochastic search in the simulated annealing approach. While the former is critical to finding "good" solutions in practical time frames, the latter is effective in avoiding cycling in the solution space. This conviction is supported by our initial success for our new approach, called *stochastic probe*, which will be reported in another paper very shortly.

Appendix A

Given a positive integer k, a *partition* of k is a set of positive integers $\{k_1, k_2, \ldots, k_m\}$ ($m \leq k$) such that $k = \sum_{i=1}^{m} k_i$. In this appendix we want to prove the following theorem:

Theorem 1: Given positive integers m and k such that $m \leq k$, any partition of k into $P = \{k_1, k_2, \ldots, k_m\}$ maximizing $\sum_{1 \leq i < j \leq m} k_i k_j$ will minimize $\sum_{1 \leq i < j \leq m} |k_i - k_j|$.

□

First we prove the following two lemmas.

Lemma 1: Let x and y be positive integers. If $x > y + 1$, then $x^2 + y^2 > (x-1)^2 + (y+1)^2$.

□

Proof: If $x > y + 1$, then $2x - 1 > 2y + 1$. Therefore $x^2 - x^2 + 2x - 1 > y^2 - y^2 + 2y + 1$, or $x^2 - (x-1)^2 > (y+1)^2 - y^2$. So we have the lemma.

□

Lemma 2: Let m and k ($m \leq k$) be positive integers and $P = \{k_1, k_2, \ldots, k_m\}$ a partition of k. Assume that there exists a pair x and y in P such that $x - y > 1$. Let $x' = x - 1$, $y' = y + 1$, and $P' = P - \{x, y\} \cup \{x', y'\}$. We have

$$\sum_{\substack{1 \leq i < j \leq m \\ k_i, k_j \in P}} |k_i - k_j| > \sum_{\substack{1 \leq i < j \leq m \\ k_i, k_j \in P'}} |k_i - k_j| \tag{1}$$

and

$$\sum_{\substack{1\le i<j\le m\\ k_i,k_j\in P}} k_i k_j > \sum_{\substack{1\le i<j\le m\\ k_i,k_j\in P'}} k_i k_j \tag{2}$$

□

Proof: We can partition P into $P = P_1 \cup \{x\} \cup P_2 \cup \{y\} \cup P_3$ where

P_1 – the set of numbers in P which are greater than or equal to x,
P_2 – the set of numbers in P which are smaller than x and greater than y,
P_3 – the set of numbers in P which are equal to or smaller than y.

Let N_1, N_2, N_3 be the cardinalities of P_1, P_2, and P_3 respectively. We have $N_1 + 1 + N_2 + 1 + N_3 = m$, and

$$\sum_{\substack{1\le i<j\le m\\ k_i,k_j\in P'}} |k_i - k_j| = \sum_{\substack{1\le i<j\le m\\ k_i,k_j\in P}} |k_i - k_j| + (N_1 - N_2 - 1 - N_3) + (-N_1 - 1 - N_2 + N_3)$$

$$= \sum_{\substack{1\le i<j\le m\\ k_i,k_j\in P}} |k_i - k_j| - 2(1 + N_2).$$

So we have Inequality 1.

Because $x > y + 1$, from Lemma 1, we have $x^2 + y^2 > x'^2 + y'^2$. Since

$$k^2 = \sum_{\substack{1\le l\le m\\ k_l\notin\{x,y\}}} k_l^2 + (x^2 + y^2) + 2\sum_{\substack{1\le i<j\le m\\ k_i,k_j\in P}} k_i k_j$$

$$= \sum_{\substack{1\le l\le m\\ k_l\notin\{x',y'\}}} k_l^2 + (x'^2 + y'^2) + 2\sum_{\substack{1\le i<j\le m\\ k_i,k_j\in P'}} k_i k_j,$$

we have Inequality 2.

□

Proof of Theorem 1:
Since partition $P = \{k_1, k_2, \ldots, k_m\}$ maximizes $\sum_{1\le i<j\le m} k_i k_j$, by Lemma 2, there can be no $x, y \in P$ such that $x > y + 1$. On the other hand, if $\max(P) - \min(P) \le 1$, then $\sum_{1\le i<j\le m} |k_i - k_j|$ must reach its smallest possible value $r(m - r)$, where r is the remainder of k/m. The theorem is thus proved.

□

Appendix B

If we already have the costs for s runs of an algorithm with random initial solutions, we can easily derive from these costs the expected minimal cost for k ($k \ll s$) runs of the algorithm with random initial solutions.

Let $L = (c_1, c_2, \ldots, c_s)$ be the list of given costs in nondecreasing order of their value. The expected minimal cost for k runs of the algorithm is

$$\sum_{i=1}^{s-k+1} p_i \cdot c_i$$

where p_i is the probability that c_i is the minimal for k costs randomly chosen from L. We can decompose p_i as $p_i = p_i^1 \cdot p_i^2$ where p_i^1 is the probability that none of the first $i-1$ costs in L is among the k chosen costs, and p_i^2 is the probability that c_i is among the k chosen costs. It can be verified that

$$p_i^1 = \prod_{j=0}^{k-1} \frac{(s-i+1)-j}{s-j}$$

and

$$p_i^2 = 1 - \prod_{j=0}^{k-1} \left(1 - \frac{1}{(s-i+1)-j}\right) = \frac{k}{s-i+1}.$$

The cost derived above is more reliable than the one obtained by simply running the algorithm k times because it is based on the information for a much larger population of costs.

References

1. F. Berman and L. Snyder, "On mapping parallel algorithms into parallel architectures", *J. Parallel Distributed Comput.* **4** (1987) 439–458.
2. V. Cerny, "A thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm", *J. Optim. Theory Appl.* **45** (1985) 41–51.
3. K. Efe, "Heuristic models of task assignment scheduling in distributed system", *IEEE Comput.* **31** (1982) 50–56.
4. C. M. Fiduccia and R. M. Mattheyses, "A linear-time heuristic for improving network partitions", *Proc. 19th Design Automation Conf.*, 1982, pp. 175–181.
5. M. R. Garey and D. S. Johnson, "Computers and interactability: A guide to the theory of NP-completeness", W. H. Freeman and Company, CA, 1979.
6. F. Glover, "Tabu search – Part 1", *ORSA J. Comput.* **1** (1989) 190–206.
7. F. Glover, "Tabu search – Part 2", *ORSA J. Comput.* **2** (1990) 4–32.
8. B. Hajek, "Cooling schedules for optimal annealing", *Math. Operat. Res.* **13** (1988) 311–329.
9. D. S. Johnson, C. R. Aragon, L. A. McGeoch, and C. Schevon, "Optimization by simulated annealing: An experimental evaluation; Part I, graph partitioning", *Operat. Res.* **37** (1989) 865–892.
10. B. W. Kernighan and S. Lin, "An efficient heuristic procedure for partitioning graphs", *Bell Syst. Tech. J.* **49** (1970) 291–307.
11. S. Kirkpatrick, C. D. Gelatt, Jr., and M. P. Vecchi, "Optimization by simulated annealing", *Science* **220** (1983) 671–680.

12. C. H. Lee, C. I. Park, and M. Kim, "Efficient algorithm for graph-partitioning problem using a problem transformation method", *Comput.-Aided Des.* **21** (1989) 611–618.
13. L. Tao, Y. C. Zhao, J. Guo, K. Thulasiraman, and M. N. S. Swamy, "An efficient tabu search algorithm for *m*-way graph partitioning ", *Proc. Supercomputing Symp.,* Fredericton, New Brunswick, June 1991, pp. 263–270.
14. J. D. Ullman, *Computational Aspects of VLSI*, Computer Science Press, 1984.