DIMACS Technical Report 95-34
August 1995

Path Optimization for Graph Partitioning Problems

by

Jonathan W. Berry [1]
DIMACS Center

Mark K. Goldberg [2]
Department of Computer Science
Rensselaer Polytechnic Institute

[1]this author did most of his work for this paper as a PhD student in the Department of Computer Science at Rensselaer Polytechnic Institute

[2]supported in part by NSF Grant #CCR-9214487.

DIMACS is a cooperative project of Rutgers University, Princeton University, AT&T Bell Laboratories and Bellcore.

DIMACS is an NSF Science and Technology Center, funded under contract STC–91–19999; and also receives support from the New Jersey Commission on Science and Technology.

ABSTRACT

This paper presents a new heuristic for graph partitioning called *Path Optimization* (*PO*), and the results of an extensive set of empirical comparisons of the new algorithm with two very well-known algorithms for partitioning: the Kernighan-Lin algorithm and simulated annealing. Our experiments are described in detail, and the results are presented in such a way as to reveal performance trends based on several variables. Sufficient trials are run to obtain 99% confidence intervals small enough to lead to a statistical ranking of the implementations for various circumstances. The results for geometric graphs, which have become a frequently-used benchmark in the evaluation of partitioning algorithms, show that *PO* holds an advantage over the others.

In addition to the main test suite described above, comparisons of *PO* to more recent partitioning approaches are also given. We present the results of comparisons of *PO* with a parallelized implementation of Goemans' and Williamson's 0.878 approximation algorithm, a flow-based heuristic due to Lang and Rao, and the multilevel algorithm of Hendrickson and Leland.

1 Introduction

A fundamental problem in graph theory is to partition the vertices of a graph into two disjoint sets of nearly equal size such that the number of edges with an endpoint in each set (cuts) is maximized or minimized. Partitioning into equal-sized sets such that the number of cuts is minimized is the *graph bisection* problem, while partitioning into two disjoint sets such that cuts are maximized is referred to as *max_cut*.

Some variations of partitioning problems, all *NP*-hard, include partitioning into many sets of bounded size, partitioning graphs with vertex and/or edge weights, and partitioning hypergraphs, among many others. These problems often remain *NP*-hard, even under certain simplifying assumptions about the input graphs.

The problems specifically addressed by algorithms in this work are *max_cut* and another *NP*-hard variation of *graph bisection* called the *min_quotient_cut* problem, described in [KPST84, LR88, LR93] and defined below. Given a graph $G = (V, E)$ and a partitioning π of V into disjoint sets S and $\bar{S}$, let $C(\pi)$ denote the number of edges cut. The *quotient cost* of π is defined as

$$\frac{C(\pi)}{min(|S|, |\bar{S}|)}.$$

The *min_quotient_cut* problem is that of finding a cut with minimum quotient cost. A similar problem is *min_ratio_cut*, in which the objective function to be minimized is

$$\frac{C(\pi)}{|S||\bar{S}|}.$$

These problems are convenient relaxations of the strict *graph bisection* problem;

unbalanced partitionings are legal, but yield poor objective function values.

Approximate bisection problems like *min_quotient_cut* have many well-documented applications. The most widely cited application in the literature is in logic design ([BHP83], [DH73], [Dun83], [FM82], [GB83], [Lei80], [KL70], [Ull84]). Given a set of logic elements and interconnections, how do we divide the set into two parts of nearly equal size such that the sum of the lengths of the wires connecting logic elements in different parts is minimized?

A more recent application is the mapping of complicated communication graphs onto parallel architectures. This problem calls for partitioning into k sets, either directly or using recursive bisection. The partitioning system *Chaco*, developed at Sandia National Laboratory by Hendrickson and Leland [HL93a] is directed towards this application and will be discussed in Section 4.

Several real-life applications of the *max_cut* problem are listed in [PT93]. A prime example is *via minimization*: given a chip layout with cells and nets already in place, the problem is to assign levels to wires and position the minimum possible number of *via*'s, or locations where wires change level, such that certain critical stretches of wire remain on the same level. The problem is modeled with a simple undirected graph and reduced to the problem of finding the *max_cut* of a contraction of this graph. The latter happens to be a planar graph, and there are polynomial-time algorithms for the *max_cut* problem on planar graphs [Had75, OD72]. However, a slight complication of the problem yields contracted graphs with the property that for any vertex v, $G - v$ is planar. The *max_cut* problem remains *NP*-hard for such cases.

2 Previous Work

There history of work on approximate graph bisection problems dates back at least thirty years. We will not attempt to chronicle this work here. A more thorough review is presented in [Ber94]. An excellent survey of results in spectral and polyhedral approaches to the *max_cut* problem is presented in [PT93]. Below, we will give a very brief summary of the most familiar approaches to graph partitioning.

The defacto benchmark algorithm for almost twenty five years has been the famous local search heuristic due to Kernighan and Lin [KL70] (*KL*). Many alternative approaches have been proposed and examined. Some have demonstrated improvements over *KL* for special classes of inputs, but it is remarkable, considering the importance of the problem and the wide variety of theoretical and experimental attention it has received, that no method has been shown to dominate this beautiful and simple idea in general.

Other partitioning heuristics include the Fiduccia-Mattheyses variation of *KL* [FM82], versions of *KL* employing graph contraction ([GB83], [Bui86], [JAMS89], [GG84], [GLR86], [HL93b]), simulated annealing (*SA*) ([KGV83],[JAMS89]), and genetic algorithms [Hol75].

Different approaches to the problem include network flow-based methods [SM86, LR88, LR93], spectral and polyhedral approaches [DH73, Bop87, RW90, PR91, PT93], and approximation algorithms [GWed, GW94, HP95].

3 Path Optimization

Path Optimization can be viewed as a variation of the hill-climbing local optimization partitioning procedure. Given an initial partitioning $\pi = (S, \bar{S})$, *PO* performs a variation of simple neighborhood search. The neighborhood of π is not limited to those partitionings obtainable by moving exactly one or two vertices. Instead of selecting and moving a small constant number of vertices as most local search methods do, Path Optimization develops variable-length sequences of adjacent vertices, then moves each vertex in the sequence to its opposing partition. We will call this operation a "flip-flop." We will refer to the change in the number of cuts resulting from this operation as the *flip_cost*. The Path Optimization (*PO*) procedure can be thought of as a form of local optimization with lookahead, as potentially many vertices may be examined before a single move occurs.

Like the *KL* and *SA* algorithms, *PO* depends on an initial partitioning generator. Most instances of the latter are randomized, so given an input graph, runs of these three algorithms may consist of improvements to an arbitrary number of initial partitionings. The result is the best overall partitioning observed during this process.

3.1 Graphs

The components to the basic *PO* algorithm for partitioning graphs are described in Figures 3.4, 3.3, 3.2, and 3.1. Let us define the *cell gain* of a vertex v ($cg(v)$) to be the number of edges that would be added to the cut if v were to move to the other partition. This definition will be formalized in Section 3.2. Fiduccia and Mattheyses use a similar definition in [FM82].

As in [FM82], bucket sorting can be used to store the vertices such that finding

Path Optimization	One iteration of the *PO* algorithm. The full algorithm consists of as many calls to this routine as time allows.
1.	Obtain an initial partitioning.
2.	Assign 0 to *side*.
3.	Repeat the following steps until there has not been an improvement in the objective function within the previous 5 paths:
4.	Call *find_path*(*side*). If the call succeeds, then flip-flop the vertices in the resulting path.
5.	Let *side* = !*side*.

Figure 3.1: The Path Optimization algorithm

the one with highest (or lowest) *cg* is a constant time operation. However *PO* relies on this information to a lesser extent than *KL* variations when determining its next move. After the initial vertex of a path has been selected by *cg*, *PO* finds subsequent path vertices by traversing adjacency lists and computing increments in the *flip_cost*.

The only parameter used to tune the *PO* algorithm is PATH_STARTS, which tells the *find_path*() routine how many times to start searching for a path before reporting failure. Since initial path vertices are selected in order of *cg*, the PATH_STARTS parameter controls the amount of non-greediness allowed in the selection of a initial vertices.[1]

For an example, consider Figure 3.5. Assuming that the PATH_STARTS param-

[1]NOTE: for minimization problems, the sequence of vertices in the "path" is not a true path in G; it consists of the vertices of two disjoint paths, one in each partition. The term "path" is retained to be consistent with the case of maximization.

find_path (*side*)	Find a sequence of vertices which begins in partition *side* and alternates partitions thereafter. The sequence is developed such that, aside from the initial vertex, the increment to the *flip_cost* due to each subsequent vertex is not unfavorable.

1. Repeat the following steps PATH_STARTS times, letting i vary from 1 to PATH_STARTS:

2. Let v be the vertex in partition *side* with the i'th best *cg* value. If the current objective function is a form of minimization, then select w such that w has the best *cg* value from among those vertices **not** adjacent to v and not in partition *side*.

3. Add v to a new path P. Then add w if this is a minimization problem.

4. Repeatedly call *select_next_cell*(P) until the path cannot be extended.

5. If the *flip_cost* of P is not unfavorable, signal success and return. Otherwise, return to step 2.

Figure 3.2: The *find_path* routine

eter is set to at least 3, the *PO* algorithm will find the optimal *max_cut*, while the *KL* (original or Fiduccia & Mattheyses version) will not.

3.2 Hypergraphs

Many applications of partitioning concern *hypergraphs* rather than graphs. A hypergraph G is a pair (V, E), where $V = \{v_0, v_1, \ldots, v_{n-1}\}$ is a set of vertices and

select_next_cell(P)	Find the next path vertex and add it to the path
1.	let *side* be the partition opposing that of the vertex most recently added to path P.
2.	If the problem is maximization, then let v be the vertex most recently added to P. If minimization, let v be the second most recently added vertex.
3.	Traverse the adjacency list of v until a neighbor w is found such that w is in partition *side*, w is not already in P, and *flip_cost_incr*(w, P) is not unfavorable.
4.	If such a w was found, add it to P, otherwise return *failure*.

Figure 3.3: The *select_next_cell* routine

$E = \{e_0, e_2, \ldots, e_{m-1}\}$ is a collection of subsets of V, called edges [2] (or hyperedges).

A modification of the *flip_cost_incr* routine from Section 3.1 will allow *PO* to partition hypergraphs. This section describes this modification. To formalize the notion of *flip_cost*, we introduce some notation. We will define the *flip_cost* for any arbitrary subset of vertices, although the *PO* algorithm will select much more specific subsets.

Definition 3.1 *Given a partitioning* $\pi = (S, \bar{S})$*, let* $loc(v)$ *denote the partition where* v *resides.*

Definition 3.2 *An edge* e *containing vertex* v *is called* type-0 critical *with respect to* v *iff* $\forall w \in e, loc(v) = loc(w)$*. Similarly,* e *is called* type-1 critical *with respect to* v *iff* $\forall w \in e, w \neq v$ *implies* $loc(v) \neq loc(w)$*.*

[2] In the VLSI literature, the vertices are often called *cells* and edges are called *nets*.

flip_cost_incr(v, P)	Determine whether on not vertex v is suitable to extend path P. This routine does not apply to hypergraphs (see Section 3.2)
1.	Let e_{nc} be the number of adjacencies between v and the vertices of P which share v's partition.
2.	Let e_c be the number of adjacencies between v and the vertices of P which do not share v's partition.
3.	Return $(cg(v) + 2(e_c - e_{nc}))$.

Figure 3.4: The *flip_cost_incr* routine

Definition 3.3 *Let $n(v)$ be the number of type-0 critical edges with respect to v; let $\bar{n}(v)$ be the number of type-1 critical edges with respect to v.*

Definition 3.4 *The* gain *of a vertex v, denoted* cg(v), *is defined to be $n(v) - \bar{n}(v)$.*

Definition 3.5 *An edge e is a* cut *with respect to a partitioning $\pi = (S, \bar{S})$ iff $\exists v, w \in e$ where $loc(v) = S$ and $loc(w) = \bar{S}$.*

Definition 3.6 *Given any set of vertices $Q \subseteq V(G)$ and a partitioning $\pi = (S, \bar{S})$, let $Q_s = Q \cap S$ and $Q_{\bar{S}} = Q \cap \bar{S}$.*

We would like to quantify the gain in cuts realized by changing the partitioning by "flip-flopping" Q, i.e., moving all vertices of Q_s into $\bar{S}$ and all vertices of $Q_{\bar{s}}$ into S.

Definition 3.7 *Let $C_0(Q)$ be defined as follows:*

$$C_0(Q) = \{e : e \in E(G) \wedge (e_S = \phi \vee \ e_{\bar{S}} = \phi) \wedge Q \cap e \neq \phi\}.$$

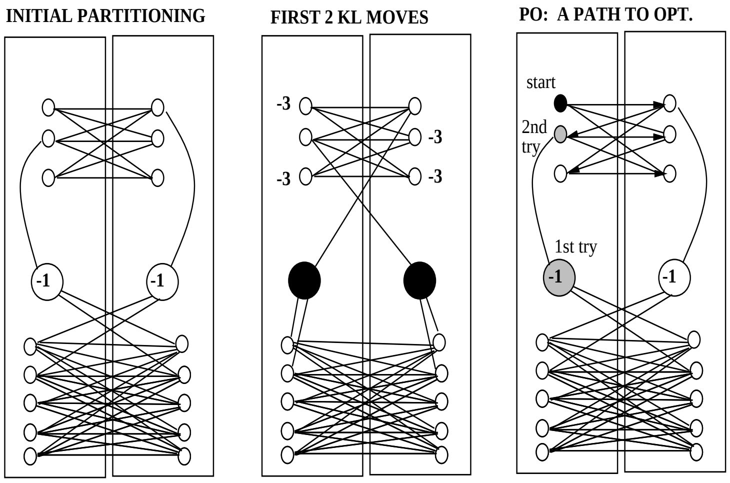

Figure 3.5: A simple example showing an operational difference between KL and PO (max_cut problem)

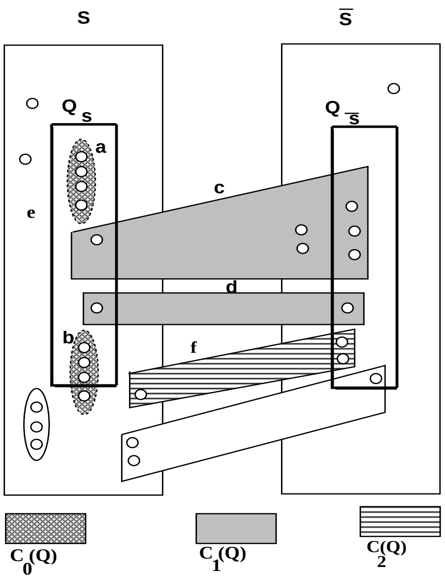

Figure 3.6: The Swap for Hypergraphs

For any vertex v in an edge $e \in C_0(Q)$, $cg(v)$ must reflect the fact that e will become a new cut if v is moved. When Q is flip-flopped, however, the vertices of Q are moved as a group. Clearly, $\sum_{v \in e} cg(v)$ will count e as a new cut edge more than once if $|e \cap Q| > 1$. For example, consider the edges a and b in Figure 3.6, which belong to $C_0(Q)$. Note that $\sum_{v \in a} cg(v) = 4$, yet if the vertices of Q are flip-flopped, a does not become a cut edge. In the case of edge b, $\sum_{v \in b} cg(v) = 4$, yet b accounts for only one more cut if the vertices in Q are flip-flopped.

Definition 3.8 *Let* $C_1(Q) = C_1(Q, S) \cup C_1(Q, \bar{S})$, *where* $C_1(Q, R) = \{e : e_R \subseteq Q \wedge\ |e_R| = 1 \wedge\ Q_{\bar{R}} \cap e \neq \phi\}$.

Edges c and d in Figure 3.6 belong to $C_1(Q)$. If the vertices of Q are flip-flopped, the sum of the cg values undercounts the actual number of edges added to the cut.

Definition 3.9 *For* $e \in C_1(Q)$, *let us define*

$$\mathcal{N}_m(e) = \begin{cases} 2 & \textit{if } |e| = 2 \\ 1 & \textit{if } |e| > 2 \end{cases}$$

For $e \in C_1(Q)$, $\mathcal{N}_m(e)$ quantifies the number of edges which are counted in $\sum_{v \in e} cg\ (v)$ as leaving the cut when the vertices of e are flip-flopped, yet which remain cut afterwards.

Definition 3.10 *Let* $C_2(Q) = C_2(Q, S) \cup C_2(Q, \bar{S})$, *where* $C_2(Q, R) = \{e : e_R \subseteq Q \wedge\ |e_R| > 1 \wedge\ e_{\bar{R}} \neq \phi\ \wedge\ Q_{\bar{R}} \cap e = \phi\}$.

This case is illustrated by edge f in Figure 3.6. The edge is uncut by the flip-flop, yet for no v does $cg(v)$ reflect this.

Definition 3.11 *Let* $q(e) = |e \cap Q|$, *and for* $e \in (C_0(Q) \cup C_2(Q))$, *let us define*

$$\mathcal{P}_m(e) = \begin{cases} 1 & \text{if } e \in C_2(Q) \\ q(e) & \text{if } q(e) = |e| \ \wedge \ e \in C_0(Q) \\ q(e) - 1 & \text{otherwise} \end{cases}$$

For an edge e, the $\mathcal{P}_m(e)$ is the component of "positive miscount" in $\sum_{v \in Q} cg(v)$ due to overestimation of the number of edges joining the cut as a result of the flip-flop.

We quantify the number of cuts gained by swapping all $v \in Q$ with the function *flip_cost*(Q), defined below

Definition 3.12

$$\text{flip\_cost}(Q) = \sum_{v \in Q} \text{cg}(v) + \sum_{e \in C_1(Q)} \mathcal{N}_m(e) - \sum_{e \in C_0(Q) \cup C_2(Q)} \mathcal{P}_m(e).$$

Note that if every edge has cardinality two, then the equation above reduces to

$$\mathit{flip\_cost}(Q) = 2(|E(Q_s, Q_{\bar{s}})| - (|E(Q_s)| + |E(Q_{\bar{s}})|)) + \sum_{v \in Q} cg(v),$$

where for any partitioning $\pi = (S, \bar{S})$, $E(S, \bar{S})$ is defined to be the set of cut edges, and $E(S)$ is defined to be the set of edges in the subgraph induced by the vertices of S.

The pseudocode for the algorithm *flip_cost_incr* is found in Figure 3.7. Following [FM82], we define the "from block" of an edge e with respect to a vertex v

```
Algorithm flip_cost_incr(v, mult)
P_m = N_m = 0
for (e in incidentEdges(v))
       e.locked[loc(v)]++
       from_locks = e.locked[loc(v)], to_locks = e.locked[!loc(v)]
       from_count = FR(v, e), to_count = TO(v, e)
       /*************** process C_0(P) **************************/
       if ((to_count == 0) && (from_locks > 1))
           P_m++
           if (from_locks == e.size)
               P_m++
       /*************** process C_2(P) **************************/
       if ((to_count > 0) && (from_count > 1) &&
               (to_locks == 0) && (from_locks == from_count))
                     P_m++
       else if ((to_count > 1)&&(to_locks == to_count)&&(from_locks == 1))
                     P_m--
       /*************** process C_1(P) **************************/
       if ((from_locks == 1)&&(to_locks > 0))
           if (to_count == 1)
                     N_m++
           if (from_count == 1)
                     N_m++
if (((v.cost + N_m - P_m) * mult) ≥ 0)
  return TRUE
else
  for (e in incidentEdges(v))
        e.locked[loc(v)]--
  return FALSE
end algorithm
```

Figure 3.7: *flip_cost_incr* pseudocode for hypergraphs

by

$$FR(v,e) = \begin{cases} e_S, & \text{if } loc(v) = S; \\ e_{\bar{S}}, & \text{otherwise.} \end{cases}$$

The "to-block" TO(v,e) is defined similarly using $\bar{S}$. Each vertex is locked when added to the current path, and each edge maintains a count of the number of its vertices which are locked.

It is shown in [FM82] that the operation of flip-flopping all of the vertices takes time $O(m)$, where m is the number of edges in G. A simple worst-case running time analysis of the PO algorithm is therefore $O(d^2sn + m)$, where d is the maximum degree of any vertex and s is the maximum size of any edge. However, the algorithm runs much faster in practice, since the average path length is typically very small. In Figures 4.9 and 4.10, the largest average path length observed was approximately 3.

A simple restriction to the PO algorithm is to select each path P such that the subgraph induced by P has no cut edges for the case of balanced minimization, and no uncut edges for the case of *max_cut*. Empirically this restricted approach has given partitionings of almost identical quality. In fact, this restricted approach is applied in Section 4.

4 Algorithm Comparisons

In this section, we describe our computational experiments and present the results. These can be divided into two main parts: an extensive comparison of the Kernighan-Lin [KL70] (Fiduccia-Mattheyses version [FM82]) (KL), simulated annealing [KGV83, JAMS89] (SA), and Path Optimization (PO) algorithms, and a more specialized evaluation of the algorithms against other approaches to partitioning such as graph contraction, flow-based methods, and the 0.878 approximation algorithm for *max_cut* [GWed].

4.1 Graph Types

The type of graphs on which we experimented, used by Johnson, *et al.* [JAMS89] and Lang and Rao [LR93], are random graphs, and randomly-generated geometric graphs. Let us denote the former by $\mathcal{R}_{n,p}$, where n is the number of vertices and p is the probability of the existence of each possible edge. For this work, we will denote the class of geometric graphs by $\mathcal{R}_{\mathcal{G}_{n,d}}$, where n is the number of vertices and d is called the *distance threshold*. The vertices of a geometric graph are distributed randomly on a unit square. Edge (v_i, v_j) exists iff the euclidean distance between v_i and v_j is less than or equal to d. Members of $\mathcal{R}_{\mathcal{G}_{n,d}}$ have also been called "unit disk graphs" in [CCJ90]. They are notable for their structured nature and offer a contrast to random graphs.

Both of these types of graphs have been used for comparisons before in the literature, and we continue the trend in order to facilitate further comparisons. Graphs of $\mathcal{R}_{\mathcal{G}_{n,d}}$ present quite a different challenge to partitioning algorithms than those of $\mathcal{R}_{n,p}$. In fact, the ranking of algorithms can be reversed when moving from $\mathcal{R}_{n,p}$ to $\mathcal{R}_{\mathcal{G}_{n,d}}$, as we will see below.

Although we have an implementation of *PO* for hypergraphs, we have not as yet run any algorithm comparisons on those inputs. Lang and Rao experimented with the MCNC benchmark circuits used in [WC89] and others, but found the graphs too small to draw any real conclusions when comparing algorithms [LR93].

4.2 Initial Partitionings

The *KL*, *SA*, and *PO* algorithms are all local search methods which accept an initial partitioning of G, then work to improve it. Let us call this process one *iteration.* A *run* of each algorithm may consist of an arbitrary number of iterations, and the result of the run is the best partitioning observed during the entire process.

Three different randomized algorithms were used as initial partitioning generators. The first is a simple algorithm which begins with empty partitions S and $\bar{S}$, and places each vertex v into S with probability 0.5 and into $\bar{S}$ with probability 0.5. We will refer to this routine as *rand.*

The second method, called the *line* heuristic, uses geometric information to split the vertex set of a geometric graph into two equal sized halves with a line of randomly chosen slope. It has been demonstrated that such initial partitionings dramatically improve the performance of *KL* and *SA* [JAMS89, LR93].

The third, which we call the *W* algorithm, is a constructive greedy procedure which starts with empty partitions, selects vertices one by one, and places them into a partition in a greedy way with respect to the objective function. The vertex selection is done by *max-diff,* defined as follows.

Let S and $\bar{S}$ be the partitions being constructed. Let $S(v)$ be the set of vertices in S that are placed before v and adjacent to v, and define $\bar{S}(v)$ similarly with respect to $\bar{S}$. For each vertex placement, let U be the set of unplaced vertices. Let $\delta(v) = |S(v)| - |\bar{S}(v)|$, the difference between the cardinalities of the two sets. Furthermore, let

$$\{v \in U : \forall w \in U, |\delta(v)| \geq |\delta(w)|\}.$$

define the set of possible candidates for the next vertex placement if the problem is *max_cut*. If the current objective function calls for minimization of the cut, the set of candidates is defined to be

$$\{v \in U : \forall w \in U, \delta(v) \geq \delta(w)\}$$

if $|S| \geq |\bar{S}|$, and

$$\{v \in U : \forall w \in U, \delta(v) \leq \delta(w)\}$$

otherwise.

In *max-diff* selection, the next vertex is drawn at random from the appropriate set of candidates. A similar vertex ordering technique was described in [CSS91]. The *W* algorithm is so named since the construction of a partitioning represents a single walk down an implicit backtracking tree (where the other branches of the tree are due to possible *non-greedy* placements. See [Ber94] for more details).

4.3 Algorithm Implementations

Our implementations of the *KL*, *SA*, and *PO* algorithms all share the exact same bucket data structure code and were implemented in C as parts of a single system by the same programmer. Our version of *KL* was tested on the set of $\mathcal{R}_{\mathcal{G}_{n,d}}$ graphs from [LR93], and it reported results comparable to those of their *KL* implementation, which in turn had been tested against that of [JAMS89]. We also obtained the code from [LR93] and ran that version of *KL* on our data sets. Our implementation of *KL* performed as well or better when running times were equalized. *SA* was not tested against any previous data sets, but our implementation is based directly on [JAMS89] and reports similar results when run on similar inputs.

Simulated annealing is a local optimization approach based on ideas arising from physics which has been used with success on several well-known discrete optimization problems. It deserves special attention since it requires careful tuning to be applied to graph partitioning successfully. The simulated annealing paradigm was developed by Kirkpatrick, Gellat, and Vecchi [KGV83]. The basic simulated annealing algorithm is a local optimization heuristic that allows "uphill" moves with a certain probability, which decreases with a *temperature* according to some *cooling schedule*. The idea is that, like the physical analogy, the "energy state," or final solution quality, is better when the cooling occurs gradually rather than suddenly. This paradigm is generic enough to be easily adaptable to a striking variety of problems. Some combinatorial applications include Graph Partitioning [JAMS89], Graph Coloring [JAMS], and Number Partitioning [JAMS], among others. However, there are several parameters inherent to the algorithm which must be set in order to tune the algorithm to a particular problem. These parameters and their settings for graph partitioning were described in Johnson *et al.* [JAMS89] and are reproduced in Figure 4.8. Generally, we use these default settings, with certain exceptions which are detailed below.

An initial annealing run at the beginning of the algorithm is used to find an appropriate starting temperature for the process. The goal is to find a temperature at which the acceptance rate for bad moves is within some *Epsilon* of *Init_Prob*.

For *max_cut* on $\mathcal{R}_{n,p}$, better results are obtained if the running time is spread over one long annealing run instead of several shorter ones. This is achieved by estimating the *Temp_Factor* for which one pass of the algorithm will take the entire allotted running time. However, for *max_cut* on $\mathcal{R}_{\mathcal{G}_{n,d}}$ and *min_quotient_cut* on both

Init_Prob	This is a target acceptance rate for bad moves used in setting the initial temperature. Default setting: 0.4.
Temp_Factor	This factor determines the rate of cooling. Default setting: .95.
Size_Factor	Determines the number of moves carried out for each temperature setting. *Epoch_Length* is defined to be $n *$ *Size_Factor*. Default setting: 16.
Min_Percent	The annealing run is "frozen" if the acceptance rate of bad moves is not greater than this and if no improvement has been obtained during the last 5 temperatures. Default setting: .02 (2%).
α	The cost of a partitioning is determined by the number of cuts times the imbalance of the partitioning times this factor. If $(S, \bar{S})$ is a partitioning, then the *cost* is given in Johnson *et al.* by $$C(S, \bar{S}) + \alpha * (\|S\| - \|\bar{S}\|)^2.$$ However, we observed better performance with the absolute difference rather than the squared difference. Default setting: 0.05.

Figure 4.8: simulated annealing parameters

graph types, the *Temp_Factor* is set as in [JAMS89] and iterations are performed until the time is up.

When the inputs are from $\mathcal{R}_{\mathcal{G}_{n,d}}$ and the starting partitioning is very good in its own right, we modify the parameters to take advantage of this and avoid "backing up." Our implementation uses *Init_Prob* = 0.02 and *Min_Percent* = .001 in this situation. These settings outperform the defaults.

Two modifications to the basic simulated annealing procedure suggested in [JAMS89] have been retained in our implementation. The first is that the sequence of potential vertex moves is chosen according to a random permutation rather than choosing individual moves at random. This change was shown to yield improvements in [JAMS89]. The second modification is that, rather than computing $e^{-\Delta/T}$ at each step, we use a lookup table of size 1000 containing values of $200\frac{\Delta}{T}$. These values approximate the value of $e^{-\Delta/T}$ to within roughly one half percent, and offer signficant speedup.

4.4 Objective Functions and Timing Information

Our implementations of the algorithms support various objective functions, including those of *max_cut* and *min_quotient_cut*. The modifications to achieve this are small.

For each algorithm, we computed the running time spent in the main loop only. The input and initialization times were not included. This gave a slight advantage to *SA*, which first makes a trial run to obtain an initial setting for its cooling ratio variable. The time was taken with the Unix *getrusage()* command. According to our experiments, the amount of work done per given time is virtually independent of the system load.

The basic algorithm comparisons shown in Figures 4.9 and 4.10 were run on a Sparc 5, Model 110 machine with rated at 78.6 SPECint92. The SPECint92 rating of a machine is a more standardized and reliable measure of speed than the MHz rating or MIPS rating. All other trials involving graphs of less than 100,000 vertices were run on Sparc 10 machines with 44.2 SPECint92 ratings. The trials involving graphs of 100,000 vertices or more were run either on Sparc 10 machines rated at 65.2 SPECint92, or RS6000 machines rated at 117 SPECint92 (comparisons are drawn only between runs on the machines with the same rating [3]).

For each variation of graph parameters, a data set of more than thirty graphs was generated if the number of vertices was less than 100,000, and each algorithm was run on all instances. Graphs with larger numbers of vertices were grouped into samples of size ten. For each algorithm, only the best solution for each graph was retained. The sample mean and standard deviation of this set of observations were then computed, as well as a 99% confidence interval for the true mean solution. For a discussion of $100(1-\alpha)\%$ confidence intervals, see [BOD86]. In standard statistical practice, a confidence interval derived from a sample size of more than thirty trials allows an appeal to The Central Limit Theorem and an argument that, with a given confidence, the true mean lies somewhere in the interval, regardless of the distribution of the individual trials. If the number of trials is less than thirty, as with our experiments with graphs of 100,000 vertices or more, the confidence interval is obtained using the Students T distribution and the assumption is made that the population of individual trials is normally distributed.

4.5 Results of Comparisons

A summary of the results of our comparisons of the *KL*, *SA*, and *PO* algorithms is presented in Figures 4.9 and 4.10. The initial partitioning generator for each algorithm is W, as indicated. Clearly, *SA* is the dominant algorithm for *max_cut* and balanced minimization problems on graphs of $\mathcal{R}_{n,p}$. Interestingly, though, for the *min_quotient_cut* problem, the rankings are reversed when the structured graphs of $\mathcal{R}_{\mathcal{G}_{n,d}}$ are considered. Here, in a class of inputs which probably more closely models

[3]Except for one case, which is documented in a footnote on Page 24.

Algorithm	mean cuts	99% conf. int.	mean q. c.	99% conf. int.	mean edges
W-PO	73.31	70.51, 76.10	0.0117	0.0113, 0.0122	48177.49
W-KL	80.53	77.24, 83.82	0.0129	0.0124, 0.0134	48177.49
W-SA	107.33	101.17, 113.49	0.0172	0.0162, 0.0182	48177.49

$\mathcal{R}_{\mathcal{G}_{12500,.0141}}$, ave. deg. $\approx$ 7.6, 91 graphs, 256 seconds per run

Algorithm	mean cuts	99% conf. int.	mean q. c.	99% conf. int.	mean edges
W-SA	11291.58	11251.02, 11332.15	1.8067	1.8002, 1.8132	47666.84
W-KL	11755.77	11725.69, 11785.86	1.8809	1.8761, 1.8857	47666.84
W-PO	12048.39	12017.07, 12079.71	1.9281	1.9231, 1.9331	47666.84

$\mathcal{R}_{12500,.00061}$, ave. deg. $\approx$ 7.6, 31 graphs, 256 seconds per run

Algorithm	mean cuts	99% conf. int.	mean q. c.	99% conf. int.	mean edges
W-SA	236434.16	236241.64, 236626.68	47.2868	47.2483, 47.3253	549969.29
W-KL	237085.55	236939.30, 237231.80	47.4171	47.3879, 47.4464	549969.29
W-PO	237631.03	237482.14, 237779.93	47.5277	47.4979, 47.5576	549969.29

$\mathcal{R}_{10000,.011}$, ave. deg. $\approx$ 110, 31 graphs, 256 seconds per run

Algorithm	mean cuts	99% conf. int.	mean q. c.	99% conf. int.	mean edges
W-PO	12723.00	12446.92, 12999.08	2.5516	2.4970, 2.6061	554804.29
W-KL	12959.03	12576.96, 13341.11	2.6162	2.5373, 2.6950	554804.29
W-SA	13540.68	12918.98, 14162.37	2.7170	2.5931, 2.8409	554804.29

$\mathcal{R}_{\mathcal{G}_{10000,.061}}$, ave. deg. $\approx$ 110, 31 graphs, 256 seconds per run

Figure 4.9: Comparisons between *SA*, *KL*, *W-PO* for the *min_quotient_cut* problem

sparse, structured inputs such as VLSI circuits and communication graphs than the others, *PO* holds a measureable edge.

The figures indicate 99% confidence intervals for solution quality based on the sample means and standard deviations of the data sets. Note that these confidence intervals overlap for the case of dense geometric graphs. More trials would tend to narrow the intervals. For the case of *min_quotient_cut* on sparse geometric graphs, 91 trials were sufficient to obtain non-overlapping intervals.

Algorithm	mean cut percentage	99% conf. int.	mean (cuts/edges)
W-PO	66.6932	66.6601, 66.7263	(32146.13 /48200.19)
W-SA	66.6634	66.6315, 66.6954	(32131.81 /48200.19)
W-KL	66.6294	66.5989, 66.6598	(32115.39 /48200.19)

$\mathcal{R}_{\mathcal{G}12500,.0141}$, ave. deg. $\approx$ 7.6, 91 graphs, 256 seconds per run

Algorithm	mean cut percentage	99% conf. int.	mean (cuts/edges)
W-SA	75.8127	75.6675, 75.9578	(36137.52 /47666.84)
W-PO	75.2885	75.2610, 75.3159	(35887.58 /47666.84)
W-KL	75.1645	75.1351, 75.1940	(35828.52 /47666.84)

$\mathcal{R}_{12500,.00061}$, ave. deg. $\approx$ 7.6, 31 graphs, 256 seconds per run

Algorithm	mean cut percentage	99% conf. int.	mean (cuts/edges)
W-SA	57.0576	57.0382, 57.0770	(313799.19 /549969.29)
W-KL	56.9048	56.8950, 56.9147	(312959.16 /549969.29)
W-PO	56.8659	56.8590, 56.8728	(312745.13 /549969.29)

$\mathcal{R}_{10000,.011}$, ave. deg. $\approx$ 110, 31 graphs, 256 seconds per run

Algorithm	mean cut percentage	99% conf. int.	mean (cuts/edges)
W-SA	56.4905	56.4751, 56.5059	(313411.55 /554804.29)
W-KL	56.4750	56.4588, 56.4912	(313325.52 /554804.29)
W-PO	56.4602	56.4453, 56.4751	(313243.29 /554804.29)

$\mathcal{R}_{\mathcal{G}10000,.061}$, ave. deg. $\approx$ 110, 31 graphs, 256 seconds per run

Figure 4.10: Comparisons between *SA*, *KL*, *W-PO* for the *max_cut* problem

4.6 Min Quotient Cut

This section presents the results of our experiments with *min_quotient_cut*, a balanced minimization problem defined on page 1. The modifications needed to switch *KL* and *PO* to solve *min_quotient_cut* are straightforward. For *SA*, the balancing is achieved through a penalty function as in [JAMS89]. Our experiments with annealing based directly on changes in quotient cut offerred no improvement in solution quality.

Algorithm	mean cuts	99% conf. int.	mean q. c.	99% conf. int.	mean edges
W-PO	73.31	70.51, 76.10	0.0117	0.0113, 0.0122	48177.49
W-KL	80.53	77.24, 83.82	0.0129	0.0124, 0.0134	48177.49
line-KL	103.99	101.18, 106.79	0.0167	0.0162, 0.0171	48177.49
line-SA	112.90	108.89, 116.91	0.0181	0.0174, 0.0187	48177.49

$\mathcal{R}_{\mathcal{G}_{12500,.0141}}$, 91 graphs, 256 seconds per run

W-PO	102.33	98.91, 105.75	0.0082	0.0079, 0.0085	95415.33
W-KL	112.58	108.87, 116.30	0.0090	0.0087, 0.0093	95415.33
line-KL	149.59	146.25, 152.94	0.0120	0.0117, 0.0122	95415.33
line-SA	151.59	147.30, 155.89	0.0121	0.0118, 0.0125	95415.33

$\mathcal{R}_{\mathcal{G}_{25000,.0099}}$, 91 graphs, 1024 seconds per run

W-PO	147.41	143.24, 151.57	0.0059	0.0057, 0.0061	191206.77
W-KL	161.43	156.08, 166.78	0.0065	0.0062, 0.0067	191206.77
W-SA	202.99	195.46, 210.51	0.0081	0.0078, 0.0084	191206.77
line-SA	212.82	208.38, 217.27	0.0085	0.0083, 0.0087	191206.77
line-KL	219.10	215.36, 222.83	0.0088	0.0086, 0.0089	191206.77

$\mathcal{R}_{\mathcal{G}_{50000,.0070}}$, 91 graphs, 4096 seconds per run

W-PO	248.90	224.28, 273.52	0.0050	0.0045, 0.0055	390723.30
line-SA	320.70	292.08, 349.32	0.0064	0.0058, 0.0070	390723.30
line-KL	331.80	310.49, 353.11	0.0066	0.0062, 0.0071	390723.30

$\mathcal{R}_{\mathcal{G}_{100000,.005}}$, 10 graphs, 28,800 seconds per run

W-PO	415.00	337.75, 492.25	0.0042	0.0034, 0.0049	766909.70
line-KL	484.10	468.18, 500.02	0.0048	0.0047, 0.0050	766909.70
line-SA	491.60	464.91, 518.29	0.0049	0.0046, 0.0052	766909.70

$\mathcal{R}_{\mathcal{G}_{200000,.0035}}$, 10 graphs, 36,000 seconds per run

Figure 4.11: Comparisons between *line-SA*, *line-KL*, *W-KL*, *W-SA*, and *W-PO*, geometric graphs of 12,500 to 200,000 vertices, average degree 7.6

As Figure 4.9 shows, there is a marked difference in the rankings of the algorithms between the $\mathcal{R}_{n,p}$ and $\mathcal{R}_{\mathcal{G}_{n,d}}$ testbeds. The *PO* algorithm holds an advantage over its competitors in the latter case. Since sparse, structured graphs probably better reflect real-life applications, we will concentrate on them for the rest of this section.

4.6.1 Increasing Graph Size

The data presented in Figure 4.11 were gathered to reveal any trends in the algorithm comparisons which might exist as graph size is increased, holding density constant. Figure 4.11 shows that the mean number of cuts produced by *W-PO* is approximately 8.9% less than that of its nearest competitor, *W-KL*, and that this approximate advantage is maintained as the number of vertices approaches 50,000.

The same figure illustrates the performance trends of *W-PO* , *line-KL*, and *line-SA* for graphs ranging from 12,500 to 200,000 vertices. The results for graphs of 100,000 and 200,000 vertices predate the linking of *W* with *KL* and *SA*; those combinations hare not yet been examined for such large graphs. The advantage in mean number of cuts maintained by *W-PO* over the others is approximately 30% for graphs having up to 50,000 vertices. Thereafter, the advantage starts to decline, though it remains greater than 14% over *line-KL* through 200,000 vertices.[4]

The %99 confidence intervals are sufficient to differentiate between *W-PO* and *W-KL* for each of the 12,500, 25,000, and 50,000 vertex test suites, though the gap between intervals is less than %3. The *W-SA* combination is not explored until graphs with 50,000 vertices, since for smaller graphs, *line-SA* lags behind *line-KL*. The improved initial partitionings help the *SA* algorithm, but not as much as

[4]The *SA* results for graphs of 200,000 vertices presented in Figure 4.11 were obtained on a machine rated at 65.2 SPECint92; those for *KL* and *PO* were gathered on a 117 SPECint92 machine.

they help *KL*. Starting with such good initial partitionings requires a low starting temperature for the *SA* algorithm (or else the good partitioning is quickly lost), perhaps limiting its effectiveness.

Note that the *W* algorithm is an excellent initial partitioning generator. For the graphs in $\mathcal{R}_{\mathcal{G}_{n,d}}$ that we examined, the *W* algorithm produces better starts than the *line* heuristic.

4.6.2 Increasing Running Time

Figure 4.12 illustrates these trends. In this figure, 99% confidence intervals are plotted graphically for *line-KL* (μ_κ), *line-SA* (μ_σ), and *W-PO* (μ_ρ). Note that the scales of the graphs in the figures are allowed to float to highlight the differences in the performance of the heuristics. The horizontal axis represents the quotient cut $(\frac{C(\pi)}{\min(|S|,|\bar{S}|)})$, while the vertical axis has no significance.

As the running time allotted to each algorithm increases, some interesting trends are revealed. The advantage of *PO* over *KL* tends to increase, and the *SA* algorithm tends to pass *KL* and begin an approach to *PO*. Still, after 50 runs of 8192 seconds each, the improved μ_σ and μ_κ do not match the sample mean of the 256 second runs of *W-PO*.

4.6.3 Comparisons with Multicommodity Flow Heuristics

In [LR93], Lang and Rao described a heuristic called *FLOW*, which is based upon the multicommodity flow approach to partitioning (see [LR88], [SM90]). The *FLOW* heuristic is used as an initial partitioning generator rather than a cleanup heuristic such as *KL* or *PO*. Lang and Rao present the results of the empirical comparison of

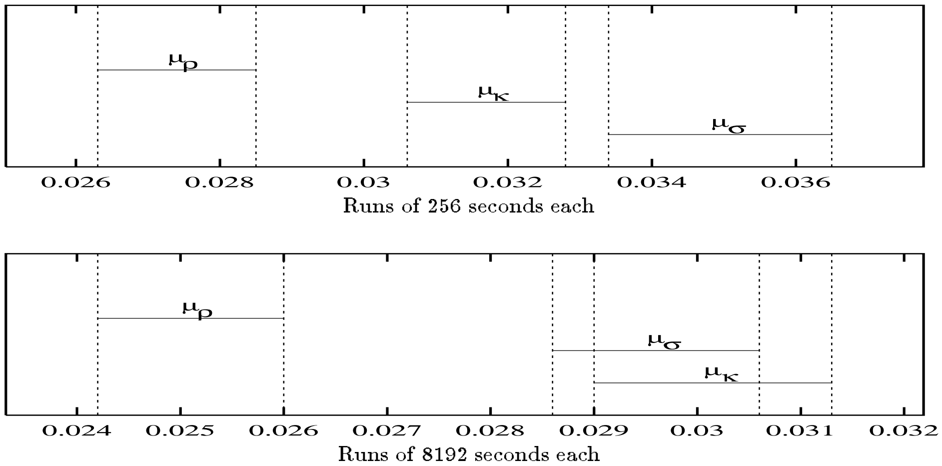

Figure 4.12: *min_quotient_cut*, 50 Geometric Graphs, n : 10,000, average degree: 10

FLOW with variations of *KL* (*FM*) applied to sparse instances of $\mathcal{R}_{n,p}$ and $\mathcal{R}_{\mathcal{G}_{n,d}}$. The authors conclude that *FLOW* is not useful for $\mathcal{R}_{n,p}$, but for $\mathcal{R}_{\mathcal{G}_{n,d}}$ it achieves better results than *line-KL* as graph size increases, provided *FLOW* is augmented with *KL* or it is given longer running time.

We made comparisons of *FLOW-KL* with *W-PO* based on the data available from [LR93]. The results of our comparisons for graphs of up to 10,000 vertices are given in Table 4.1. The *FLOW-KL* column refers to the quotient cut found by first applying *FLOW*, then cleaning up the solution with *KL*. Note that there are no iterations of this process; the majority of the running time is spent in the single execution of *FLOW*. The latter must solve many global shortest path problems and then compute a minimum spanning tree.

For graphs of these sizes, there seems to be no clear winner. Although the average quotient cuts of *FLOW-KL* are slightly better, *W-PO* produced the best quotient cut five out of twenty times. In [LR93], *FLOW-KL* was run on one graph

Size	Ave. Deg.	#graphs	line-*KL*	*FLOW-KL*	*W-PO*	#times *W-PO* best
1000	10.94	10	.1175	.1134	.1154	1
3000	11.46	5	.0796	.0726	.0750	2
10000	12.02	5	.0448	.0406	.0412	2

Table 4.1: *min_quotient_cut* Testbed of [LR93]

of 100,000 vertices and average degree 13.7. After 3 days of running on a 36 MHz silicon graphics machine, it produced a quotient cut of .014 ($\approx$ 700 cuts). In a run of similar duration on the same graph, the best achievable by *W-PO* was .019 ($\approx$ 950 cuts), and the best by *line-KL* was .020 ($\approx$ 1000 cuts). Unfortunately, *FLOW-PO* has not been explored. If an applications developer has large amounts of time available, the *FLOW* heuristic is an excellent initial partitioning generator.

4.6.4 Comparisons with Graph Contraction Heuristics

vertices	time	*W-PO*	*line-KL*	*line-SA*	*ML*
12,500	4sec	104.97,133.16	139.14,155.77	139.08,159.70	100.96,122.98
25,000	8sec	158.79,193.27	194.31,218.60	190.59,218.25	158.37,194.08
50,000	17sec	282.49,354.48	275.52,308.74	275.40,311.57	184.54,214.49
100,000	39sec	350.83,869.37	388.12,548.68	390.38,508.82	285.01,345.79
200,000	90sec	961.58,1572.62	558.25,679.75	570.13,692.47	NA

Table 4.2: Very short runs: (*ML* is the multilevel algorithm of [HL93b]) Expected # Cuts (same graph classes as Figure 4.11

The idea of partitioning large graphs by performing a series of graph contractions has been explored in [GB83], [Bui86], [JAMS89], and [HL93b]. In [Bui86] and [JAMS89], empirical evidence is presented indicating that a contracted version of

KL can improve the algorithm both in speed and quality if the input graphs are very sparse. Bui [Bui86] finds an advantage for regular graphs of a special class only if the degree is four or less.

In [HL93b], Hendrickson and Leland give a multilevel algorithm which uses weighted intermediate graphs to preserve good partitionings as the graph is uncontracted. The contractions are obtained by finding maximal matchings and identifying endpoints of matching edges. After the resultant graph is partitioned using the spectral method of [HL92], the original graph is restored through a series of uncontractions, with *KL* (*FM*) occasionally cleaning the partitioning. Results are presented indicating that for bisection of large, sparse graphs, this algorithm performs significantly better than spectral partitioning alone.

Using *Chaco* [HL93a], a partitioning system due to Hendrickson and Leland which implements several spectral partitioning methods and the multilevel algorithm described above, we were able to make limited comparisons with *W-PO*, *line-KL*, and *line-SA*. The algorithms were run on a subset of the suite of graphs from Figure 4.11, and the results are presented in Table 4.2.[5] The intended application for *Chaco* is the mapping of parallel computations, where speed is obviously extremely important. The multilevel algorithm is very fast, while the heuristics, especially *W-PO*, require some time to work well. The advantage of the multilevel algorithm increases with graph size for these short running times. However, its expected solution quality falls short of the longer runs of *W-PO* (see Figure 4.11). A natural way to extend the running time of the multilevel algorithm would be to randomize the contraction process and perform many iterations. In future work, we hope to experiment with

[5] Unfortunately, the 200,000 vertex testbed could not be completed for the multilevel algorithm due to run-time errors in the version of Chaco we obtained.

this possibility, and test the multilevel algorithm with longer runs and *PO* as a cleanup routine.

4.7 Max Cut

The results of Johnson *et al.* [JAMS89], which concerned Graph Bisection, a minimization problem, suggested that simulated annealing was slightly better than *KL* for $\mathcal{R}_{n,p}$ and clearly worse for $\mathcal{R}_{\mathcal{G}_{n,d}}$. Our results show that this is not the case in general, even for minimization of the cut for $\mathcal{R}_{\mathcal{G}_{n,d}}$ (see Section 4.6). In fact, for the case of *max_cut*, *SA* was the overall winner for both $\mathcal{R}_{n,p}$ and $\mathcal{R}_{\mathcal{G}_{n,d}}$. For sparser graphs of $\mathcal{R}_{\mathcal{G}_{n,d}}$, *PO* holds a very slight advantage that is quickly lost as the number of vertices grows.

4.7.1 Comparisons Between Heuristics

Table 4.13 shows the results of comparisons on larger geometric graphs. These results give another indication that the *W* algorithm enerates good initial partitionings for local search heuristics. Note that when *W* is used before each run of *KL*, the average best *KL* partitioning improves enough to pass that of *rand-SA* for the graph testbeds of 12500 and 25000 vertices. For these sparser graphs, the sample mean partitioning of *W-PO* holds a very slight advantage over *W-KL*. Note however that the trend of increasing graph size favors *SA*. Even without the benefit of the *W* algorithm, *SA* is able to dominate *W-PO* and *W-KL* as graph size approaches 50,000 vertices.

Unlike the case of *min_quotient_cut* on sparse graphs of $\mathcal{R}_{\mathcal{G}_{n,d}}$ (see Figure 4.12),

Algorithm	graphs	sample mean	99% conf. int.	cuts/edges
W-PO	30	66.6680	66.6271, 66.7089	32135.3000/48202.1333
W-KL	30	66.6166	66.5820, 66.6512	32110.5333/48202.1333
SA	30	66.4695	66.3142, 66.6247	32039.9333/48202.1333
KL	30	66.1503	66.1178, 66.1828	31885.7667/48202.1333

max_cut, $\mathcal{R}_{\mathcal{G}_{12500,.0141}}$ (average degree 7.6), runs of 256 seconds

Algorithm	graphs	sample mean	99% conf. int.	cuts/edges
W-PO	30	66.7413	66.7149, 66.7676	63712.4333/95461.9667
W-KL	30	66.6966	66.6697, 66.7234	63669.7333/95461.9667
SA	30	66.7169	66.5203, 66.9135	63689.5333/95461.9667
KL	30	66.2049	66.1794, 66.2304	63200.4333/95461.9667

max_cut, $\mathcal{R}_{\mathcal{G}_{25000,.0099}}$ (average degree 7.6), runs of 1024 seconds

Algorithm	graphs	sample mean	99% conf. int.	cuts/edges
SA	30	66.8402	66.6406 ,67.0399	127876.9667/191318.0666
W-PO	30	66.7283	66.7042 ,66.7523	127663.0667/191318.0666
W-KL	30	66.6817	66.6583 ,66.7050	127573.9333/191318.0666
KL	30	66.1744	66.1496 ,66.1992	126603.5000/191318.0666

max_cut, $\mathcal{R}_{\mathcal{G}_{50000,.0070}}$ (average degree 7.6), runs of 4096 seconds

Figure 4.13: Max Cut Comparisons on Large, Sparse Geometric Graphs

the trends as running time allowed increases do not foretell a change in algorithm rankings. Figure 4.14 shows that as the running time increases from approximately five minutes to over two hours, the relative positioning of the algorithms is unchanged. In this figure, the horizontal axis depicts the percentage of edges cut, while the vertical axis has no significance.

4.7.2 Comparisons Between Heuristics and the Best Approximation Algorithm

Recently, Goemans and Williamson ([GW94],[GWed]) constructed an approximation algorithm which delivers solutions to *max_cut* with a performance expectation of at least .87856 and also computes an upper bound which does not exceed the op-

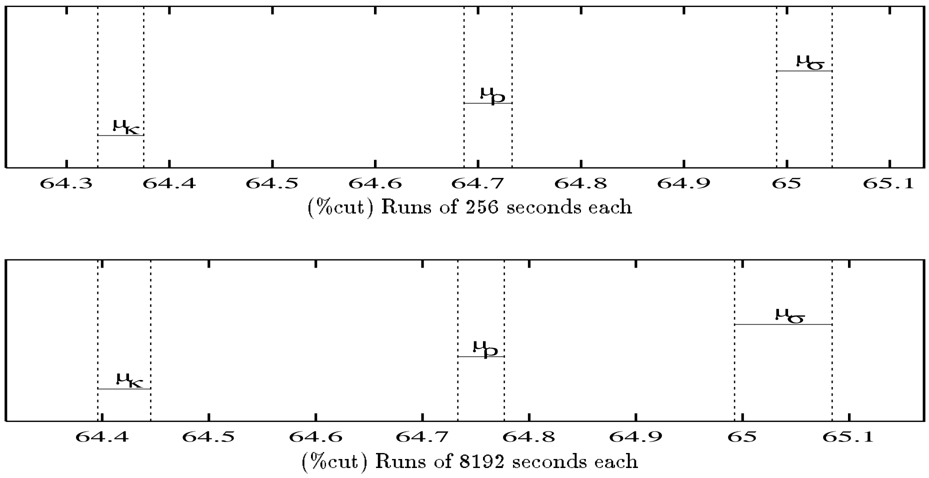

Figure 4.14: *max_cut* : 99% Confidence Intervals, 50 Geometric Graphs, n : 10,000, ave deg: 10

n	edges	upper bound	*W-SA*	*W-PO*	*W-KL*	GW_{cm5}
6999	35319	27865	25830	25432	25468	24964
7999	39824	31510	29114	28714	28750	28255
8000	39951	31585	29193	28764	28766	28307
6999	34927	27600	25540	25163	25135	24725
6000	30138	23770	22020	21724	21728	21311
5000	24938	19690	18252	17997	17893	17671

Table 4.3: Comparisons with known upper bounds for *max_cut.* These are sparse random graphs $\mathcal{R}_{n,\frac{10}{n}}$, data from [HP95]. The three heuristics are run for *15 seconds* of Sparcstation-20 time each. The approximation algorithm results were obtained in a matter of minutes on the CM-5.

timal value by more than a factor of $\frac{1}{0.87856}$. The algorithm is based on semidefinite programming [6]

Goemans and Williamson [GWed] cite the complexity of solving the semidefinite

[6] A semidefinite program is an optimization problem of a linear function of a symmetric, positive semidefinite matrix subject to linear equality constraints. See [GWed] for further references.

n	edges	upper bound	*W-SA*	*W-PO*	*W-KL*	GW_{cm5}
6999	35319	27865	25894	25520	25506	24964
7999	39824	31510	29268	28880	28815	28255
8000	39951	31585	29351	28877	28865	28307
6999	34927	27600	25665	25281	25284	24725
6000	30138	23770	22107	21839	21761	21311
5000	24938	19690	18323	18075	18034	17671

Table 4.4: Comparisons with known upper bounds for *max_cut*. These are sparse random graphs $\mathcal{R}_{n,\frac{10}{n}}$, data from [HP95]. The three heuristics are run for *one hour* of Sparcstation-20 time each. The approximation algorithm results were obtained in a matter of minutes on the CM-5.

program to within an additive error of ϵ as $O(n^{3.5}(\log W_{tot} + \log \frac{1}{\epsilon}))$, where W_{tot} is the sum of the weights of the edges. This complexity is not practical for large instances of the problem; however, Homer and Peinado give a parallelized version of *GW* in [HP95]. Guaranteed polynomial time convergence is traded for a method with faster practical running time and an optimal parallelization. Using this method, it is possible to solve instances of random graphs with up to 8000 vertices in about 45 minutes on a 32-node Connection Machine (CM-5).

n	edges	upper bound	*W-SA*	*W-PO*	*W-KL*	GW_{cm5}
5029	14805	10965	10520	10460	10467	10356
4927	14410	10680	10264	10218	10217	10076

Table 4.5: Comparisons with known upper bounds for *max_cut*. These are sparse geometric graphs, data from [HP95]. The three heuristics are run for *15 seconds* of Sparcstation-20 time each. The approximation algorithm results were obtained in a matter of minutes on the CM-5.

Tables 4.3, 4.4, 4.5, and 4.6 present a comparison between partitionings ob-

n	edges	upper bound	*W-SA*	*W-PO*	*W-KL*	GW_{cm5}
5029	14805	10965	10589	10496	10499	10356
4927	14410	10680	10325	10221	10223	10076

Table 4.6: Comparisons with known upper bounds for *max_cut*. These are sparse geometric graphs, data from [HP95]. The three heuristics are run for *one hour* of Sparcstation-20 time each. The approximation algorithm results were obtained in a matter of minutes on the CM-5.

tained by the heuristics with Homer and Peinado's parallelized implementation of The Goemans-Williamson algorithm on a 32 node Connection Machine CM-5. The heuristics consistently come closer to the upper bound than the approximation algorithm, with *W-SA* leading all competitors.

5 Conclusion and Future Work

The *PO* results we presented were obtained by a straightforward variation of the algorithm. Clearly though, many variations on this theme are possible. We examined versions of *KL* which incorporate the ideas of *PO* into a *KL* pass and which use *PO* as a cleanup routine after each pass, respectively. Neither offered a significant improvement over standard *KL*. We also examined, without finding improvement, a version of *PO* which selects the next vertex such that it is adjacent to *any* previous path vertices in the appropriate partition. This yields an underlying tree (or pair of trees) rather than a path.

Any procedure which attempts to develop vertex subsets of equal but not bounded size in each partition such that the vertices within each subset are tightly-connected, yet the subsets are loosely connected, retains the flavor of *PO* (for minimization in this example).

The Path Optimization algorithm is defined for hypergraphs, but the experiments have been limited to binary graphs. Certainly, future work is to explore this area.

Finally, the W algorithm has been shown to provide excellent starting partitionings when sparse, structured graphs are considered. In [Ber94], a probabilistic-greedy (PG) variation of the greedy W algorithm is described and shown to outperform W in certain cases. The $PG - PO$ combination has yet to be explored.

References

[Ber94] J. W. Berry. *Path Optimization for Graph Partitioning Problems: A Case Study of Near Greedy Analysis.* PhD thesis, Rensselaer Polytechnic Institute, Dec 1994.

[BHP83] M. Burstein, S. J. Hong, and R. Pelavin. Hierarchical VLSI layout: Simultaneous placement and wiring of gate arrays. In *Proceedings IFIP VLSI-83*, August 1983.

[BOD86] Bruce L. Bowerman, Richard T. O'Connell, and David A. Dickey. *Linear Statistical Models: An Applied Approach.* Duxbury Press, Boston, Massachusetts, 1986.

[Bop87] R. B. Boppana. Eigenvalues and graph bisection: An average case. In *Proceedings of the 28th Symposium on the Foundations of Computer Science*, pages 280–285, 1987.

[Bui86] Thang N. Bui. *Graph Bisection Algorithms.* PhD thesis, Massachussetts Institute of Technology, 1986.

[CCJ90] B.N. Clark, C.J. Colbourn, and D.S. Johnson. Unit disk graphs. *Discrete Mathematics*, 86:165–177, 1990.

[CSS91] L.H. Clark, F. Shahrokhi, and L.A. Szekely. A linear time algorithm for graph partition problems. Technical report, University of New Mexico, Department of Mathematics and Statistics, 1991.

[DH73] William E. Donath and A. J. Hoffman. Lower bounds for the partitioning of graphs. *IBM Journal of Research and Development*, 17:420–425, 1973.

[Dun83] A. E. Dunlop. Automatic layout of gate arrays. In *Proceedings ISCAS-83*, pages 1245–1248, May 1983.

[FM82] Charles M. Fiduccia and R. M. Mattheyses. A linear-time heuristic for improving network partitions. In *Proceedings of the 19^{th} IEEE Design Automation Conference*, pages 175–181, 1982.

[GB83] M. K. Goldberg and M. Burstein. Heuristic improvement technique for bisection of VLSI networks. In *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers (ICCD '83)*, pages 122–125, 1983.

[GG84] M. K. Goldberg and R. Gardner. On the minimal cut problem. In *Progress in Graph Theory*, pages 295–305. Academic Press, 1984.

[GLR86] M. K. Goldberg, S. Lath, and J. W. Roberts. Heuristics for the graph bisection problem. Technical Report 86-8, Rensselaer Polytechnic Institute, Department of Computer Science, 1986.

[GW94] M. Goemans and D. Williamson. .878-approximation algorithms for max cut and max 2sat. In *Proceedings of the 26^{nd} ACM Symposium on Theory of Computing*, 1994.

[GWed] M. Goemans and D. Williamson. Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming. *Journal of the ACM*, submitted.

[Had75] F.O. Hadlock. Finding a maximum cut of a planar graph in polynomial time. *SIAM Journal of Computing*, 4:221–225, 1975.

[HL92] B. Hendrickson and R. Leland. An improved spectral graph partitioning algorithm for mapping parallel computations. Technical Report SAND92-1460, Sandia National Laboratories, 1992.

[HL93a] B. Hendrickson and R. Leland. The chaco users guide, version 1.0. Technical Report SAND93-2339, Sandia National Laboratories, 1993.

[HL93b] B. Hendrickson and R. Leland. A multilevel algorithm for partitioning graphs. Technical Report SAND93-1301, Sandia National Laboratories, 1993.

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

[HP95] S. Homer and M. Peinado. A highly parallel algorithm to approximate maxcut on distributed memory architechtures. In *International Parallel Programming Symposium*, 1995.

[JAMS] D. S. Johnson, C. R. Aragon, L. A. McGeoch, and C. Schevon. Optimization by simulated annealing: An experimental evaluation, part II (graph coloring and number partitioning). To appear in Operations Research.

[JAMS89] D. S. Johnson, C. R. Aragon, L. A. McGeoch, and C. Schevon. Optimization by simulated annealing; part I, graph partitioning. *Operations Research*, 37:865–892, 1989.

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

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

[KPST84] P. Klein, S. Plotkin, C. Stein, and E. Tardos. Faster approximations algorithms for the unit capacity concurrent flow problem with applications to routing and finding sparse cuts. In *Proceedings of the 22nd annual Symposium on Theory of Computing*, pages 310–321, 1984.

[Lei80] C. Leiserson. Area-efficient graph layout (for VLSI). In *Proceedings of the 21^{st} Annual IEEE Symposium on the Foundations of Computer Science*, pages 270–281, 1980.

[LR88] F. Thomas Leighton and Satish Rao. An approximate max-flow min-cut theorem for uniform multicommodity flow problems with applications to approximation algorithms. In *Proceedings of the 29^{th} IEEE Symposium on the Foundations of Computer Science*, pages 422–431, 1988.

[LR93] K. Lang and S. Rao. Finding near-optimal cuts: an empirical evaluation. In *Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, January 1993.

[OD72] G.I. Orlova and Y.G. Dorfman. Finding the maximal cut in a graph. *Engrg. Cybernetics*, pages 502–504, 1972.

[PR91] S. Poljak and F. Rendl. Solving the max-cut problem using eigenvalues. Technical Report 199, Technische Universitat Graz, Institute fur Mathematik, 1991.

[PT93] S. Poljak and Z. Tuza. The max-cut problem - a survey. *in* Special Year on Combinatorial Optimization*, DIMACS series in Discrete Mathematics and Theoretical Computer Science, L. Lovasz and P. Seymour, editors*, page (to be published), 1993.

[RW90] F. Rendl and H. Wolkowicz. A projection technique for partitioning the nodes fo a graph. Technical Report 169, Technische Universitat Graz, Institute fur Mathematik, 1990.

[SM86] F. Shahrokhi and D.W. Matula. The maximum concurrent flow problem and sparsest cuts. Technical report, Southern Methodist University, March 1986.

[SM90] F. Shahrokhi and D.W. Matula. The maximum concurrent flow problem. *Journal of the ACM*, 37:318–334, 1990.

[Ull84] J. Ullman. *Computational Aspects of VLSI.* Computer Science Press, Inc., Rockville, Maryland, 1984.

[WC89] Y. C. Wei and C.K. Cheng. Towards an efficient hierarchical designs by ratio cut partitioning. In *Proceedings of the IEEE International Conference on computer-aided design*, pages 298–301, 1989.