HYPERCUBE EMBEDDING HEURISTICS: AN EVALUATION [1]

Woei-Kae Chen
Department of Electrical and Computer Engineering
North Carolina State University
Raleigh, NC 27695-7911

Matthias F.M. Stallmann and *Edward F. Gehringer*
Department of Computer Science
North Carolina State University
Raleigh, NC 27695-8206

May, 1990
(to appear in *International Journal of Parallel Programming*)

volume 18, number 6 (1989)
pages 505-549

[1] This research is partially supported by the Office of Naval Research under contract N00014-88-K-0555, which is gratefully acknowledged

HYPERCUBE EMBEDDING HEURISTICS: AN EVALUATION

WOEI-KAE CHEN , MATTHIAS F.M. STALLMANN AND EDWARD F. GEHRINGER

Abstract. The hypercube embedding problem, a restricted version of the general mapping problem, is the problem of mapping a set of communicating processes to a hypercube multiprocessor. The goal is to find a mapping that minimizes the length of the paths between communicating processes. Unfortunately the hypercube embedding problem has been shown to be NP-hard. Thus many heuristics have been proposed for hypercube embedding. This paper evaluates several hypercube embedding heuristics, including simulated annealing, local search, greedy, and recursive mincut bipartitioning. In addition to known heuristics, we propose a new greedy heuristic, a new Kernighan-Lin style heuristic, and some new features to enhance local search. We then assess variations of these strategies (e.g., different neighborhood structures) and combinations of them (e.g., greedy as a front end of iterative improvement heuristics). The asymptotic running times of the heuristics are given, based on efficient implementations using a priority-queue data structure.

Key Words. mapping problem, hypercube embedding, greedy heuristics, local search, simulated annealing, priority queues

1. Introduction and Problem Description.

The communication pattern of a parallel algorithm can be represented by a communication graph. To execute the algorithm efficiently on a multiprocessor, one attempts to allocate communicating processes to adjacent processors insofar as possible, so that the communication overhead is minimized. This problem is known as the *mapping problem* [4]. More formally, a set of communicating processes defines a *communication graph* (or *task graph*) in which vertices represent processes, and an edge between process v and process w means that communication between v and w occurs during execution. An *interconnection graph* (or *target graph*) defines interconnections among the processors on which processes will execute. Each vertex represents a processor and an edge represents a physical link between processors.

In recent years, hypercube multiprocessors [8, 24, 37] have become popular, due to their high regularity, low average distance, and comparatively low construction cost. This motivates the study of the hypercube embedding (mapping) problem, which is the special case of the mapping problem where the interconnection graph is a hypercube.

Of course, it may not be possible to allocate *all* adjacent processes to adjacent processors. For example, a triangle can never be embedded in a hypercube exactly. Even where an exact embedding is possible in a hypercube of unlimited size (as is the case for tree-structured communication graphs [41]), the number of available processors may not be large enough to allow it. Thus, a performance metric (the objective function) has to be defined. We will now formally define the hypercube embedding problem, beginning with a definition of a hypercube.

A *k-cube* is an undirected graph $H = (V_h, E_h)$ consisting of $n = 2^k$ vertices labeled from 0 to $n-1$, such that there is an edge between any two vertices if and only if the binary representation of their labels differs by exactly one bit.

According to the above definition, each vertex in a k-cube has exactly k neighbors (degree k); there are a total of $nk/2$ edges; and the average distance between pairs of nodes is $kn/2(n-1) \approx k/2$. We shall use H to denote an arbitrary hypercube and H_k for a k-cube. Also, a graph G is said to be *k-cubical* if G is isomorphic to a subgraph of H_k.

An embedding f of a graph $G = (V, E)$ to a k-cube $H_k = (V_h, E_h)$ is a one-to-one function $f : V \mapsto V_h$. For a mapping function f, the dilation cost of an edge $\{v, w\} \in E$ is $d(f(v), f(w))$, i.e., the Hamming distance between $f(v)$ and $f(w)$. The expansion cost of the embedding f is $n/|V|$, i.e., reciprocal of the processor utilization. Clearly, a graph G is k-cubical if and only if every edge of G can be embedded in a k-cube in dilation 1.

Our graph-theoretic notation is standard. When describing the communication graph we use n to denote the number of vertices, m to denote the number of edges, $deg(v)$ for the degree of vertex v, and $A(v)$ for the set of neighbors of vertex v, i.e. $\{w \mid \{v, w\} \in E\}$. In reference to a specific mapping f, $cn(v, d)$ denotes v's neighbor along cube dimension d, the vertex w for which $f(v)$ and $f(w)$ agree at all but the dth bit (f is understood from the context). The notation $f \oplus (v, w)$ means a mapping that is the same as f, but with the mappings of two vertices $f(v)$ and $f(w)$ interchanged.

1.1. Importance of Problem. When commercial hypercube multiprocessors were first introduced in 1985, the mapping problem was a major concern. The store-and-forward message-passing strategy was used, meaning that messages between non-adjacent processors were stored in memory at intermediate processors until such time as they could be forwarded toward their destination. Message-transmission time was essentially proportional to path length, so it was important to obtain a low-cost mapping.

Beginning in 1987, the second generation of commercial hypercubes broke the linear relationship between path length and transmission time by using more sophisticated "virtual cut-through" routing networks, such as the Intel iPSC/2's Direct-Connect^{TM} routing [26]. The iPSC/2 is able to route a message to the most distant processor in a 128-node network in only 10% more time than it takes to reach an adjacent node. With the new generation of machines, the mapping problem temporarily lost its importance.

We should not expect this situation to persist. Interconnection networks in second-generation hypercubes tend to have more capacity than their processors are capable of utilizing. Routing times will be comparatively uniform only as long as networks remain uncongested; when a network becomes congested, delays will grow with increasing path length. Since the interconnection structure is an underutilized resource, and one whose cost grows as $N \log N$, designers can be expected to modify their architectures to make more effective use of it. For example, multiple processors can be placed at each hypercube vertex. The DASH multiprocessor project [32] has placed 4 processors at each vertex. This follows the lead of the Connection Machine 2 [40], which has 16 processors per vertex (although the hypercube embedding problem is not relevant to its unusual architecture). In these systems, we can presume that link utilization will be driven high enough for the mapping problem to matter again, despite the fact that store-and-forward overhead has been removed.

For synchronous algorithms, the quality of a mapping f is usually evaluated by its expansion cost and maximum dilation, $\max_{\{v,w\} \in E} d(f(v), f(w))$ [25]. If the maximum dilation is a constant (independent of the problem size), the maximum communication delay can be treated as a constant. Thus the complexity of the algorithm does not increase as a result of the mapping. On the other hand, if the maximum dilation is not constant, the overall complexity can be k times greater on a k-cube, or even worse due to congestion in network links.

For asynchronous algorithms, communication overhead is usually formulated as a quadratic assignment problem. That is, given a communication graph $G = (V, E)$ and a mapping function f, the total communication cost is

$$\sum_{\{v,w\}\in E} \mathit{weight}(v,w) \cdot d(f(v), f(w)).$$

If all the edges in the communication graph have the same weight, the average dilation, i.e. $(1/|E|) \cdot \sum_{\{v,w\}\in E} d(f(v), f(w))$, is used as the performance metric.

The hypercube embedding problem can thus be defined as follows: Given a communication graph $G = (V, E)$ and a k-cube, find a mapping function f such that the objective function (e.g., maximum dilation, total communication cost, or average dilation) is minimized.

Unfortunately, the problem of identifying whether a given graph is k-cubical has been shown to be NP-complete [16, 42]. In fact, even when the graph is a tree, the problem is still NP-complete. Since an optimal hypercube embedding algorithm should be able to obtain the exact embedding if the given graph is a subgraph of a k-cube (i.e., be able to answer whether a given graph is a subgraph of a k-cube), the hypercube embedding problem is NP-hard, no matter which objective function is used.

Many heuristics have been proposed for the general mapping problem [3, 9, 20, 21, 34, 36, 38, 39] and the hypercube embedding problem [14, 17, 18, 30, 31, 41]. This paper focuses on the performance of different heuristics for the hypercube embedding problem. Most previous experimental work has used a version of the quadratic assignment problem, either simplified (e.g., [9, 14]) or complicated (e.g., [17, 31]), as an objective function. We have chosen to adopt average dilation as our standard performance metric. We wanted to avoid having our results depend on the mechanism for assigning weights to edges, another source of variability. Average dilation offers good insight into the communication delay induced by various mappings. Mappings with lower average dilations will also encounter less queuing delay due to link congestion. It is likely that other objective functions would produce similar results, insofar as the relative performance of the various strategies is concerned. In our experimental data we often report the total *cost* of an embedding, $\sum_{\{v,w\}\in E} d(f(v), f(w))$, in place of or in addition to the average dilation (large whole numbers are easier to compare than fractions).

1.2. Previous Theoretical Work. Garey and Graham [22] analyzed critical subgraphs and characterizations of hypercubes. They posed a number of open questions, including, "Does there exist a characterization of cubical graphs leading to an effective procedure for recognizing these graphs?" Wagner and Corneil [42] showed that the embedding of general graphs to hypercubes is NP-complete, and even embedding general trees in fixed-size hypercubes is NP-complete. It should be noted that the problem of whether an n-vertex graph G is k-cubical, where $k = \lceil \log n \rceil$, has been shown to be NP-complete only in cases where G consists of many disjoint connected components [16]. It appears that the complexity of the special case addressed by the experiments reported here (embedding a connected graph with exactly 2^k vertices onto H_k) is still open.

A number of researchers have studied the hypercube embedding for regular structures, such as hypercubes themselves [6], grids [5, 11, 12], trees [41, 43], and binary trees [7, 33, 41, 43]. In particular, Bhat showed that testing whether a given graph is exactly a hypercube can be done in $O(n \log n)$ time and Chan [12] showed that all 2D grids can be embedded in their optimal cubes (the smallest cube that has at least as many nodes as the grid) in dilation 2.

It is known that some binary trees cannot be embedded into their optimal cubes with dilation 1 (e.g., complete binary trees; see [43]). Afrati, Papadimitriou, and Papageorgiou [1] described a divide-and-conquer algorithm that gives dilation-1 embeddings of a k-cubical tree (k is the smallest k possible) in a hypercube of dimension at most k^2. In the case of binary trees, this algorithm embeds an n-node binary tree in a hypercube of at most $O(n^{1.71})$ nodes (based on the fact that binary trees can always be divided into two subtrees each with 1/3 and 2/3 of the nodes). Wagner [41] improved this result by showing that any binary tree can be embedded in an $O(n \log n)$-node hypercube in dilation 1.

Bhatt, Chung, Leighton, and Rosenberg first showed that an arbitrary binary tree can be embedded in a hypercube with $O(1)$ dilation, $O(1)$ expansion, and also $O(1)$ congestion [7]. The constant factor of this embedding is too large to make it of practical interest. Monien and Sudbrough [33] improved the result by giving an embedding of dilation 3 and expansion $O(1)$ and an embedding of dilation 5 and expansion 1. Finally, the embedding of a k-ary tree of height d was shown by Wu [43] to have a dilation $2 \log k$ on a $((d-1)\log k + 1)$ cube.

1.3. Previous Experimental Work. A well-known early result due to Bokhari [9] proposed a local-search algorithm with pairwise exchange for mapping communication graphs to a Finite Element Machine (FEM, "eight-nearest neighbor" interconnection). Bokhari adopted the cardinality model (i.e., maximizing the number of edges of the communication graph that are mapped with dilation 1) as the performance metric and tested about 20 structural problems of 9 to 49 nodes for FEM's of size 4×4 to 7×7. To avoid local-optima traps, probabilistic jumps were used in the local search to improve the performance.

Lee and Aggarwal [31] formulated a set of new objective functions which accurately quantify communication overhead for different applications (e.g., asynchronous communication, synchronous communication, and parallel image-processing model). A greedy heuristic in combination with a local search (pairwise exchange) was also proposed to solve the mapping problem. The algorithm was tested for 9 problem graphs on hypercubes of 8 and 16 nodes respectively.

A simulated annealing algorithm was studied and reported by Ramanujam, Ercal, and Sadayappan [36]. To formulate the communication overhead, a load-imbalance factor was taken into account as part of the objective function. Two strategies, namely simulated annealing with scaling and simulated annealing with exchange, were investigated to prevent the load-imbalance factor from trapping the process at local optima. Simulations were done for 5 structured and 3 random graphs with 144 to 602 vertices.

Ercal, Ramanujam, and Sadayappan further proposed an efficient recursive mapping strategy for hypercubes [17] based on repeated application of the Kernighan-Lin graph partitioning heuristic [28]. This algorithm was compared with simulated annealing under the same set of test graphs in [36]. Results showed that this algorithm obtained costs slightly worse than simulated annealing, but the cost difference was always less than 10% and the computation time of their recursive strategy was several orders of magnitude less.

A processor-and-link assignment algorithm using simulated annealing was developed by Bollinger and Midkiff [10]. Since each link in the multiprocessor may be used by several processes (causing traffic congestion on the link), the objective function considers communication costs and the load on each link. The algorithm employs two optimization phases. Process annealing assigns processes to processors (processor assignment) and then connection annealing further reduces the communication cost by routing traffic over data paths (link assignment). The performance of the algorithm was evaluated by mapping hypercubes with 8 to 512 nodes onto themselves and mapping binary trees to hypercubes. Simulation showed that this simulated annealing algorithm was able to consistently map hypercubes of size ≤ 128 perfectly.

Recently André, Pazat, and Priol compared the performance of several different mapping algorithms for the hypercube embedding problem [2]. They adopted the quadratic assignment problem as the objective function and compared four different heuristics including Bokhari's algorithm [9], Chen's algorithm [14], a simulated annealing algorithm, and what they call the "friendly greedy" algorithm [35]. The comparison was based on mapping 4×4 grids to 16-node hypercubes and mapping a parallel ray-tracing algorithm to 16- and 32-node hypercubes. For each graph and each algorithm, 100 experiments were performed and algorithms were compared by their average cost.

1.4. Summary of Contributions. The purpose of our study is to evaluate the performance of a variety of different heuristics for hypercube embedding on a variety of different communication graphs. We chose the 7-cube as our primary target graph and all of our communication graphs had exactly 128 vertices. Differences among heuristics with smaller cubes as targets were not as striking (e.g., the 7-cube was the smallest target on which the superiority of simulated annealing for random graphs became clear). Choosing a larger cube would have severely limited the number of tests we were able to do. We did follow up our more interesting findings on larger cubes. Many interesting observations emerged; some of these need to be pursued with more extensive testing, and, where possible, confidence intervals can be determined for parameters of interest.

The main contributions of our study are the following.

- Comparisons of 12 heuristics or combinations of heuristics on 7 different types of communication graphs (a total of 61 different graphs were used).
- Extensive testing and evaluation of each individual heuristic to obtain a com-

petitive implementation.

- Significant improvements in runtime and/or solution quality for several heuristics.
- A new fast (linear-time) greedy heuristic that obtains significantly better than random solutions.
- An adaptation of the Kernighan-Lin graph partitioning heuristic with results that are competitive with simulated annealing.
- The use of flat moves, transformations that neither increase nor decrease cost, to significantly improve the solution quality of local-search heuristics.
- Tests on random geometric graphs, a class of graphs that exhibits more structure than random graphs, but less than other classes, to observe the effect of limited structure on the efficacy of the heuristics (random geometric graphs were used to test graph partitioning heuristics by Johnson et al. [27]).

The rest of our paper is organized as follows. Section 2 gives a description of each of the heuristics we tested, and our implementations. Section 3 describes our experimental methodology. Section 4 gives a detailed account of our results. Section 5 gives some conclusions and Section 6 gives suggestions for future work.

2. The Heuristics. The following is a description of each of the heuristics we implemented, with some indication of asymptotic running time and overall performance characteristics. The running time of most heuristics on sparse graphs is improved significantly by the use of an efficient priority-queue implementation discussed in the first subsection. Since most communication graphs are likely to be sparse and since the effort to improve mappings of dense graphs may not pay off anyway (for sufficiently dense graphs, even random solutions are likely to be close to optimal), the priority-queue implementation may well be worth the effort.

Table 1 gives a summary of all the heuristics we tested, showing their asymptotic running times for graphs of arbitrary density (m = number of edges, n = number of vertices) and the expected running time for graphs having average degree $\log n$, the same as the hypercube. Local search, Kernighan-Lin, and simulated annealing are also referred to as *iterative improvement* heuristics, because existing solutions are repeatedly improved by applying transformations. For these iterative improvement heuristics l is the expected number of iterations of the outer loop, whose value can only be determined experimentally, since it depends on the rate at which the heuristic converges to a local optimum.

- For local-search heuristics, l appeared to grow roughly as $O(n \log n)$, which is the order of the expected difference in cost between a random starting solution and the final local optimum (the average amount of improvement per iteration was a small constant).
- For Kernighan-Lin l appeared to grow as $O(\log n)$.
- For simulated annealing l was dependent on the settings of various parameters.

Also shown are actual average runtimes on a Sun 3/260, the machine used to obtain our experimental results. The runtimes shown are for hypercubes of dimension 7; we checked the asymptotic runtimes with similar experiments on cubes of dimensions 6, 8, and 9. These runtimes are, in the case of local search and Kernighan-Lin, for the unenhanced versions of these heuristics. Runtimes reported later are larger due to the addition of flat moves or random uphill moves.

TABLE 1
Asymptotic running times of various heuristics

Heuristic	Type	Time (general)	Time (cube)	CPU sec.
G	greedy	$O(mn)$	$O(n^2 \log n)$	0.86
SG	greedy	$O(m+n)$	$O(n \log n)$	0.03
LS	local search	$O(mn + l(m + n \log n))$	$O(n^2(\log n)^2)$	4.87
LSC	local search	$O(m \log n + l(m/n + \log n))$	$O(n(\log n)^2)$	0.25
KL	Kernighan-Lin	$O(lmn)$	$O(n^2(\log n)^2)$	14.62
RMB	graph partition	$O(l(m \log n))$	$O(ln(\log n)^2)$	0.92
SAC	simulated annealing	$O(l(m \log n))$	$O(ln(\log n)^2)$	175.17

(m = number of edges, n = number of vertices, and l = number of iterations in outer loop)

Based on 5 runs on each of 10 random graphs (128 vertices, an average of 448 edges), mapped onto cubes of dimension 7 — each heuristic was tested on the same set of graphs and all iterative improvement heuristics, i.e. local search, Kernighan-Lin, and simulated annealing used the same 50 random initial solutions.

2.1. Efficient Data Structures. Many of the heuristics described below make use of an efficient priority-queue implementation called a *bucket list* (used by Fiduccia and Mattheyses [19] in their implementation of the Kernighan-Lin heuristic for graph partitioning). The general setting is one in which there is a finite set of items X and an integer value $gain(x)$ in some limited range for each $x \in X$. Figure 1 shows the bucket list as used in our implementation of Chen's greedy heuristic (G); each item is a pair (v, h), where v is a communication graph vertex and h is a hypercube vertex. An item can also be a single communication graph vertex (in the simple greedy heuristic SG), a pair of vertices to be swapped (in LS and KL), or a pair (v, d), where v is a vertex and d is a dimension along which v is to be swapped (in LSC). The priority queue supports the operations

- *Insert*(x) (put x on the queue),
- *Delete*(x) (remove x from the queue),
- *Max*$()$ (return the item with largest gain), and
- *Change*(x) (revise x's position in the queue in accordance with the current value of $gain(x)$).

The following lemma is a restatement of the time bound found in [19], presented in more general terms.

LEMMA 2.1. *Let Q_I be the number of times the priority-queue operations* Insert, Delete, *and* Change *are called, let Q_M be the number of calls to* Max, *let M be the maximum* gain *of any item, and let δ be the maximum amount by which the* gain *of*

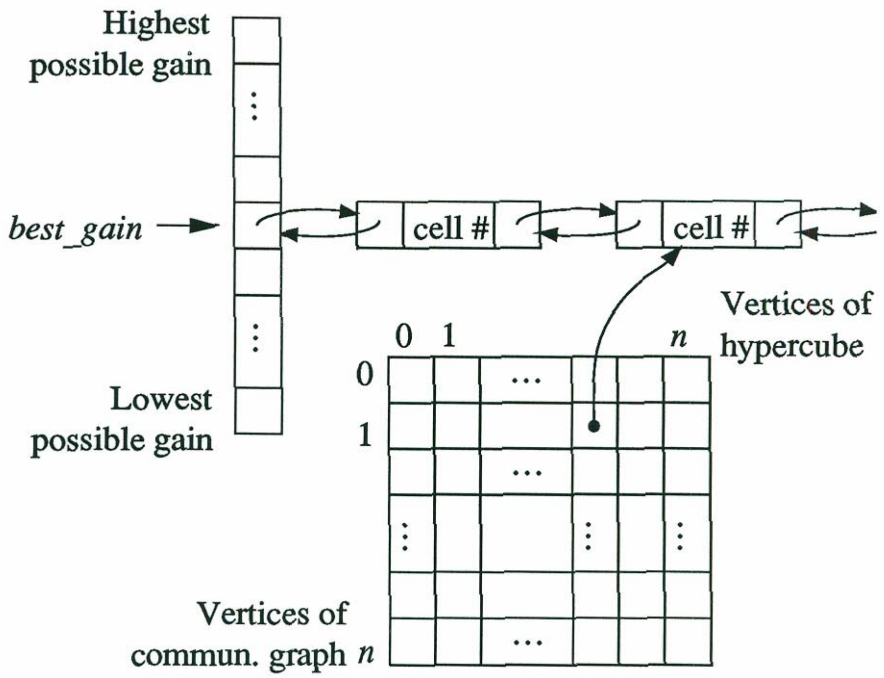


FIG. 1. *"Bucket list" priority queue*

any item may increase between two successive Delete *operations. Then, assuming all insertions occur at the beginning, the priority queue can be implemented so that the total time for all operations is $O(Q_I + Q_M\delta + M)$.*

Proof. The priority queue is stored as an array of M distinct buckets, where *bucket*[i] is a linked list of all items whose current gain is i. If every item has a pointer to its position in the appropriate list, the operation *Change* is simply a matter of moving an item from one bucket to another and can be done in constant time. *Insert* and *Delete* can also be done in constant time. To facilitate *Max* we maintain an auxiliary variable *best_gain*, the value of the largest i for which *bucket*[i] is non-empty; *best_gain* is updated whenever the *gain* of an item becomes larger than the current value of *best_gain* as the result of *Insert* or *Change*. After a *Delete*, *best_gain* may have to be decreased (if the bucket containing the item of highest gain was emptied) by scanning for the next non-empty bucket. However, the scan for the next non-empty bucket may be deferred until the next *Max* operation. If the time spent scanning is ignored, *Max* can be done in constant time.

It is easy to see that the total time spent scanning for the next non-empty bucket during *Max* is proportional to the number of buckets, plus the sum over all *Max* operations of the increase in *best_gain* since the previous *Max*. Except for the increases due to the initial insertions, which account for a total of at most M, *best_gain* cannot increase in value unless the *gain* of an individual item increases by at least the same amount. Thus the time spent scanning is $O(Q_M\delta + M)$. □

Some comments are in order. First, the restriction that all insertions take place at the beginning is really not a restriction. We can always choose a sufficiently small gain to represent the fact that an item is not in the queue, insert all items initially with that value, and use *Change* to simulate all subsequent insertions and deletions. Second, the bound given by the lemma is overly pessimistic if the number of *Max* operations exceeds the number of deletions. This is not an issue in our applications of the lemma. Finally, if δ is large, as may be the case when these heuristics are adapted to the quadratic assignment model, an ordinary priority queue (heap) can be used to achieve a time bound of $O((Q_I + Q_M + M) \log s)$, where s is the maximum number of items in the queue. The effect on our reported time bounds is at most an additional $\log n$ factor.

2.2. Greedy Heuristics. Greedy heuristics for hypercube embedding are known to be efficient, easy to implement, stable (predictable), and capable of mapping regular structures well (e.g., many greedy heuristics generate the optimal solution when the communication graph is a cube). In applications where a solution needs to be generated quickly and coding effort is at a premium, a greedy heuristic may be the best choice.

In our experimental evaluation, greedy heuristics are also used as a front end to iterative improvement algorithms to generate better initial solutions. Aside from simulated annealing, this appears to be the combination that gives lowest-cost solutions.

The generic form of a greedy heuristic for hypercube embedding is given in Figure 2. Running time and solution quality vary with the sophistication of the gain function. We implemented one greedy heuristic of moderate sophistication, namely that of Chen [13, 14], referred to as G, and one very simple one, referred to as SG. Chen's heuristic appears to be typical in both runtime and solution quality of the other greedy heuristics found in the literature. We did some experiments with the heuristic of Lee and Aggarwal [31] to verify this claim. Both heuristics as implemented have an asymptotic running time of $O(mn)$, although our implementation of Lee and Aggarwal's appears to be slightly faster. Solution quality for Chen's heuristic is uniformly better, except on perfect cubes, where both obtain optimal solutions. The gap between the two (both runtime and solution quality) increases with increasing dimension. Experiments done by André et al. on smaller graphs had a similar outcome [2]. Note that our implementation of Chen's heuristic, due to the use of Lemma 2.1 is more efficient than reported in [13]. The same tricks can easily be applied to the other greedy heuristics, and we did use them in our implementation of Lee and Aggarwal's.

In Chen's heuristic

$$gain(v, h) = \sum_{w \in A(v), w \notin V} (\log n - d(h, f(w))).$$

In other words, *gain* is a measure of how much better position h is than the worst conceivable mapping for v. If a bucket list is used to keep track of *gain*, the first statement in the loop of Figure 2 is implemented by the operation *Max*(). Updates to *gain* are accomplished by the procedure in Figure 3.

There are n *Max* operations, n^2 *Deletes*, and $O(mn)$ *Changes*. The gain of any item increases by at most $\log n - 1$ during any iteration. Note also that the maximum

```
$V :=$ communication graph vertices
$H :=$ hypercube nodes
initialize $gain : V \times H \mapsto [0 \ldots C]$
**repeat**
    choose a pair $(v^*, h^*)$ with $v^* \in V$, $h^* \in H$, such that
        $gain(v^*, h^*) = \max_{v \in V, h \in H}\{gain(v, h)\}$
    map $v^*$ to $h^*$
    $V := V - \{v^*\}$; $H := H - \{h^*\}$
    update values of $gain$
**until** $V = \emptyset$
```

FIG. 2. *Generic greedy heuristic for hypercube embedding*

```
$Delete(v^*, h^*)$
**for** $v \in V$ **do** $Delete(v, h^*)$ **end do**
**for** $h \in H$ **do** $Delete(v^*, h)$ **end do**
**for** $v \in A(v^*) \cap V$ **do**
    **for** $h \in H$ **do**
        $gain(v, h) := gain(v, h) + \log n - d(h, f(v))$
        $Change(v, h)$
    **end do**
**end do**
```

FIG. 3. *Updating gain in Chen's heuristic*

possible gain of an item is $(n-1)(\log n - 1)$. We can therefore apply Lemma 2.1 with Q_I in $O(mn)$, $Q_M = n$, M in $O(n \log n)$, and δ in $O(\log n)$ to obtain a time bound of $O(mn)$.

Since the level of sophistication attained by most of the greedy heuristics reported in the literature is at the cost of a time bound that is worse than quadratic in the input size, we decided to see whether similar solution quality could be achieved by a linear-time greedy heuristic. One way to achieve linear time is to make the gain independent of h and to use a predetermined sequence of hypercube nodes to guide the mapping.

The Gray code sequence (see [23] for details) is a natural candidate for an ordering of hypercube vertices. It has the property that each vertex in the sequence is adjacent to both its immediate predecessor and its immediate successor. Also, vertices that are close to each other in Gray code order are likely to be close in Hamming distance. In every iteration the simple greedy heuristic chooses a vertex that has a maximum number of neighbors already mapped. Figure 4 gives an overview. If a bucket list is used to maintain $|A(v) \cap V'|$ for each v and the communication graph is in adjacency-list format, the overall time is $O(m+n)$ (dominated by $O(m)$ *Change* operations; $\delta = 1$ in this case — see Lemma 2.1). For best results the buckets should be implemented as last-in first-out lists. This favors vertices whose neighbors were mapped to more recent nodes in the Gray code sequence and are thus more likely to be close to the current node.

```
$V' := \emptyset$                /* vertices already mapped */
**for** $h \in H$ (in Gray code order) **do**
    choose $v^* \in V - V'$ so that $|A(v^*) \cap V'|$ is maximized
    map $v^*$ to $h$
    $V' := V' \cup \{v^*\}$
    **for** $v \in A(v^*)$ **do**
        increment $|A(v) \cap V'|$ (update data structure)
    **end do**
**end do**
```

FIG. 4. *Simple greedy heuristic*

While SG, the simple greedy heuristic, is by far the fastest heuristic we tested, its solution quality is also the worst. The solutions obtained by SG on random graphs had costs that were roughly halfway between random solutions and the best solutions obtained by simulated annealing. In combination with LSC, the fast local search described in the next section, SG is still faster than any other heuristic (including LSC by itself) and solution costs are competitive with other heuristics. We suspect that with some additional sophistication obtainable at the cost of only an additional $O(\log n)$ factor in the runtime, the simple greedy heuristic can be made competitive with other greedy heuristics.

2.3. Local Search. Local search is a general term for heuristics that repeatedly improve the quality of a solution by applying local transformations. Two choices must be made in implementing a local-search heuristic. The first of these is the choice of *neighborhood*, the set of transformations that may be applied to the current solution in order to obtain a new one. The simplest approach is to transform a solution by exchanging, or swapping, the hypercube nodes to which two vertices are mapped. We consider two possibilities. In the *all-swaps* neighborhood, all $n(n-1)/2$ possible exchanges are considered. The *cube-neighbors* neighborhood reduces the *neighborhood size*, the number of possible transformations, to $(n \log n)/2$ by allowing swaps only between vertices that are mapped to adjacent hypercube nodes. Not surprisingly, the cube-neighbors approach, while significantly quicker, also gives solutions that are not quite as good as those of the all-swaps approach.

The second choice is the discipline by which a transformation is chosen. Three possibilities are *first descent* — consider the the swaps in arbitrary order and choose the first one that improves the current solution, *steepest descent* — choose the swap that gives the greatest possible improvement over the current solution, and *random descent* — choose at random among swaps that improve the current solution. Random descent is actually a special case of simulated annealing, described later.

We initially implemented a simple all-swaps, first-descent local-search heuristic. This implementation, while significantly slower than the ones reported in our results (its worst-case running time was $O(lmn)$), uses no non-trivial data structures and is therefore easy to implement. It also uses less additional space than LS, $O(m)$ versus $O(n^2)$, a significant consideration when mapping sparse graphs onto cubes of dimension 9 or higher. In our experiments, the simple implementation obtained slightly better solutions than those reported for LS.

The two implementations of local search in our experiments are LS, an all-swaps steepest-descent implementation, and LSC, a cube-neighbors steepest-descent implementation.

LS keeps the possible swaps on a bucket list, choosing the best swap at each stage. The gain of a swap is the cost decrease that results if the swap is performed. Before we give details of the time bound, it should be noted that the bucket list strategy can also be adapted to a first-descent or random-descent discipline. In this case the gain is 1 if the swap leads to lower cost and 0 otherwise.

The gains of all the swaps can be initialized in $O(mn)$ time: for each pair of vertices x, y compute the gain of the swap x, y by computing distances from x and y to their neighbors both before and after the swap. The gain of any single swap is in the range $\pm(n-1)(\log n - 1)$, so the maximum gain M is $O(n \log n)$. Whenever a swap v, w is done, it is sufficient to update the gains for the following pairs:

- v, x and w, x for all x: a recomputation of the gain of each of these swaps can be done in time $O(deg(v) + deg(x))$ (and $O(deg(w) + deg(x))$) for a total time of $O(n\tilde{d})$, where $\tilde{d}$ is the expected degree of a vertex (or neighbor) involved in a swap; this step also requires $O(n)$ *Change* operations.
- x, y for $x \in A(v) \cup A(w)$, all y: these gains need only be adjusted according to

the new distance between x and v or w; time is dominated by $O((deg(v) + deg(w))n)$ *Change* operations, assuming unit cost to compute hypercube distances.

In the formula of Lemma 2.1, Q_I is dominated by $O(ln\tilde{d})$ *Change* operations, $Q_M = l$, $\delta = M = n \log n$. Our experiments confirm that the expected degree of vertices involved in swaps is the same as for vertices in general, that is $\tilde{d} \approx m/n$.

The overall running time of LS is therefore $O(mn + l(m + n \log n))$. Recall that l is the number of swaps before a local optimum is reached. From our experiments l appears to grow as $O(n \log n)$. Using a greedy initial solution (see Section 2.2) reduces l significantly and also improves solution quality. The observed running time with a random starting solution for graphs having $m = n \log n$, where $n = 64$, 128, 256, and 512, was consistent with our theoretical analysis, somewhat worse than $O(n^2)$.

LSC runs in expected time $O(l(m/n + \log n))$, plus $O(m \log n)$ for initialization, with the growth rate of l being similar to that observed for LS. On the graphs used to test asymptotic runtime, both time bounds simplify to $O(n(\log n)^2)$ (assuming $l \approx n \log n$), which is consistent with the observed results. Whether the speedup in runtime makes up for the loss in solution quality is application dependent.

The time bound for LSC is obtained with a generalization of a trick used by Kernighan and Lin [28]. Let $\gamma(v, d)$ be the decrease in cost that would result from changing dimension d of the mapping of vertex v (without changing the mapping of any other vertices); that is, $\gamma(v, d)$ is the number of vertices in $A(v)$ whose dth bit in the mapping is different from that of v, minus the number whose dth bit is the same. Then $gain(v, d)$, the decrease in cost from performing a swap of v with $w = cn(v, d)$, is $\gamma(v, d) + \gamma(w, d) - 2 \times adj(v, w)$, where $adj(v, w)$ is 1 if v and w are adjacent in the communication graph, 0 otherwise. Given a specific mapping, the values of $\gamma(v, d)$ for all v and d can be initialized in time $O(m \log n)$. When v and $w = cn(v, d)$ are swapped, the values of $\gamma(v, d)$, $\gamma(w, d)$, and $\gamma(x, d)$ for all $x \in A(v) \cup A(w)$ are changed. This changes the value of *gain* for all these vertices and their hypercube neighbors along dimension d. Since v and w each have new hypercube neighbors along all other dimensions as well, the values $gain(v, d')$ and $gain(cn(v, d'), d')$ must change for each dimension d'. The same goes for $gain(w, d')$ and $gain(cn(w, d'), d')$.

The total number of items whose gain is changed is $O(deg(v) + deg(w) + \log n)$. Note that the maximum amount by which $gain(x, d')$ for any x, d' can change during the swap is $O(deg(x))$; the maximum value of $gain(x, d')$ is $deg(x)$, the minimum value is $-deg(x)$. The running time after initialization, using Lemma 2.1 is $O(l(\tilde{d} + \log n))$, where $\tilde{d}$ is the average degree of vertices involved in swaps. As in the bound for the all-swaps steepest-descent heuristic, we use the experimental observation that $\tilde{d} \approx m/n$.

The communication-graph adjacency matrix for computing $gain(v, d)$ (which would require $O(n^2)$ time and space) is not necessary. We only need to know for each vertex v and dimension d whether $cn(v, d) \in A(v)$. For this purpose it suffices to maintain an $O(n \log n)$ size array adj, where $adj[v][d]$ is 1 if $cn(v, d) \in A(v)$, 0 otherwise. If we assume two cube addresses can be checked for adjacency in constant time, adj can easily be updated as γ is updated during a swap. Otherwise, the time bound for LSC needs to be multiplied by a $\log n$ factor. Our experiments actually used the explicit adjacency matrix; the extra space was not a critical factor.

We added an additional wrinkle to the traditional local-search approach. During our early experiments we discovered that many of the available swaps neither increase nor decrease solution cost. A traditional local-search heuristic will halt when no further improvement is possible, i.e. when all available swaps either increase cost or keep it the same. We allow a certain number of *flat moves*, moves that keep cost the same, when the current configuration cannot be improved. The number of successive flat moves is controlled by an input parameter *maxflatmoves*. Increasing this parameter increases the running time (because more swaps are made overall), but also improves solution quality, up to a point.

LS and LSC have the advantage that flat moves are less costly than for simple local search. In a straightforward implementation of local search, each flat move requires a scan of all possible swaps to make sure none of them improve cost before a flat move is chosen. In LS and LSC, because we always choose the best possible move at each stage, a flat move is no more costly than any other move. Figure 5 shows the tradeoff between runtime and solution quality when flat moves are added to LS (based on 20 runs for one graph, either random or geometric). The pattern is similar for LSC (with lower runtimes). In our experiments we chose 80 as the standard setting of *maxflatmoves* for LS and 40 for LSC — beyond that point the improvement in solution cost (for random and geometric graphs) was less than the standard error of the data.

The importance of flat moves is questionable, of course, when more general cost functions, such as the quadratic assignment model, are used. However, it is still possible to allow a limited number of moves each of which increases cost by less than some small threshold. We suspect that the effect of such a strategy on more general cost functions will be similar to the effect that flat moves had here on local-search heuristics.

2.4. Kernighan-Lin. The Kernighan-Lin heuristic for graph partitioning [28] is a variant of local search. We use the term "Kernighan-Lin" to denote any heuristic that, during a single *stage*, tries out a sequence of best moves (swaps in this case) and chooses as its initial solution for the next stage the point of lowest cost in the sequence. Our implementation of KL, the Kernighan-Lin heuristic for hypercube embedding, is shown in Figure 6. To allow escapes from local optima and thus offer better competition with simulated annealing in solution quality, we allowed a certain number of randomized uphill stages, stages during which no improvement occurs (this includes stages which neither increase nor decrease cost). The randomization could be refined so that each move during a stage is weighted according to how close to the initial point it is (moves that go farther uphill would receive less weight, analogous to simulated annealing). This refinement did not improve solution quality for random graphs; our reported results are for the unrefined version. Completely randomized uphill moves appear to be a more powerful mechanism for escaping local optima than flat moves, particularly when mapping communication graphs that are almost hypercubes. Since the Kernighan-Lin approach computes a whole sequence of moves during every stage, it is a simple matter to choose a random position in that sequence as the destination for a "jump."

Figure 7 shows the increase in runtime and decrease in mapping cost obtained

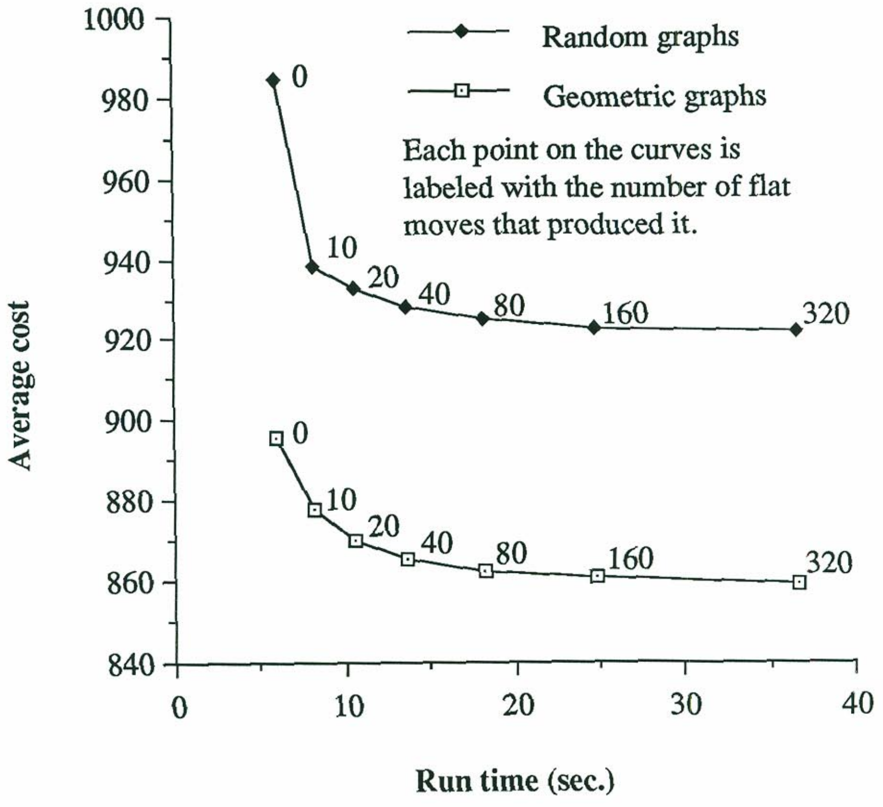


FIG. 5. *Adding flat moves to LS*

```
$f :=$ an initial mapping (e.g. random or greedy)
$best\_f := f$
$best\_cost :=$ cost of $f$
$not\_better := 0$
**repeat**
    /* this is the beginning of a *stage* */
    initialize a bucket list and insert items $v, w$ into it
        (one item for each possible swap;
         let $gain(v, w) =$ decrease in cost from swapping $v, w$)
    /* "try out" a sequence of swaps, using each item once */
    $V' :=$ communication graph vertices
    $cost[0] :=$ cost of $f$
    **for** $i = 1$ **to** $n/2$ **do**
        $v, w := Max()$
        $swap[i] := v, w$
        $cost[i] := cost[i-1] - gain(v, w)$
        $V' := V' - \{v, w\}$
        $Delete(v, w)$
        **for** $x \in V'$ **do** $Delete(v, x)$; $Delete(w, x)$ **end do**
        **for** $x \in A(v) \cup A(w)$ **do**
            **for** $y \in V'$, $y \neq x$ **do** update $gain(x, y)$; $Change(x, y)$ **end do**
        **end do**
    **end do**
    /* find best cost along the way and perform swaps on current
        mapping to get there */
    $cost[i^*] := \min_{0 \leq i \leq n \log n} cost[i]$
    **if** $cost[i^*] < best\_cost$ **then**
        $not\_better := 0$
        **for** $i = 1$ **to** $i^*$ **do** $f := f \oplus swap[i]$ **end do**
        $best\_f := f$
        $best\_cost := cost[i^*]$
     **else** /* no improvement: try a random uphill or flat move */
        $not\_better := not\_better + 1$
        $i^* :=$ a random number from 1 to $n/2$
        **for** $i = 1$ **to** $i^*$ **do** $f := f \oplus swap[i]$ **end do**
    **endif**
**until** $not\_better \leq max\_uphill\_moves$
report $best\_cost$ and $best\_f$
```

FIG. 6. *Kernighan-Lin heuristic for hypercube embedding*

by increasing *maxuphillmoves*, the parameter that controls the number of consecutive random jumps allowed (based on 20 runs for one graph, random or geometric). We arbitrarily chose the value of 20 for *maxuphillmoves*, partly because this value gave solutions that were competitive with simulated annealing on all classes of graphs, with significantly better runtime. In most cases, increasing *maxuphillmoves* will lead to even better solutions, but also significantly increased runtime.

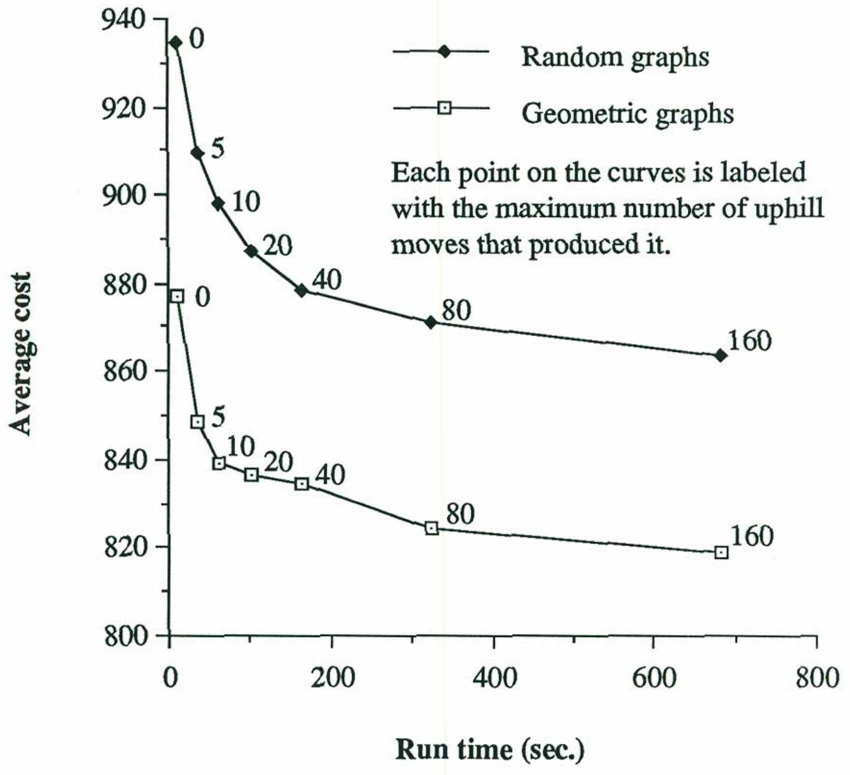


FIG. 7. *Effect of random jumps on the Kernighan-Lin heuristic*

Using an analysis similar to that of LS, we can show that the asymptotic time bound for KL is O(lmn), where l is the number of stages. The value of l is also much smaller than that for local search (average values for l when mapping a random graph to a cube of dimension 6, 7, 8, or 9 were 6, 11, 14, and 19, respectively — based on 20 runs; these results were obtained without uphill moves; l is, of course, influenced by the value of *maxuphillmoves*). In practice, KL is slower than LS but also gives significantly better solutions.

2.5. Recursive Mincut Bipartitioning. Proposed by Ercal et al. [17], *recursive mincut bipartitioning* is an interesting blend of greedy strategies and those based on local search. A fundamental insight, which is also the basis of much of the theoretical work on hypercube embedding (see, e.g. [7, 33]), is that good embeddings can be obtained by repeatedly subdividing the communication graph into pieces of roughly equal size so as to minimize the number of edges between pieces. The effect is to minimize the number of edges that might map to long paths because their endpoints are in different subcubes.

A detailed overview of RMB is given in Figure 8. If a graph $G = (V, E)$ having $n = 2^k$ vertices is to be mapped to the k-cube H_k, we split G into two subgraphs $G_0 = (V_0, E_0)$ and $G_1 = (V_1, E_1)$ so that the number of edges between V_0 and V_1 is minimized. Then we recursively map each of G_0 and G_1 into disjoint H_{k-1}, subcubes of H_k. A graph with one vertex is mapped trivially to a 0-cube. If the mappings of G_0 and G_1 are completely unaware of each other, the edges between V_0 and V_1 may end up being mapped in an undesirable way. One way to optimize the mapping of these *cross edges* would be to try out various rotations of the subcube mappings relative to each other. The number of possible combinations makes this approach too expensive. Actually the RMB heuristic maps G_0 first and then completes the mapping of G_1 using information about the mapping of G_0, i.e. the cross edges are counted when computing the cost of a subpartition of G_1. When the current subgraph is partitioned and the dth bit of f is chosen for its vertices, vertices in other subgraphs whose dth bit has already been chosen are taken into account.

We used the Fiduccia and Mattheyses [19] implementation of the Kernighan-Lin heuristic [28] to do the graph partitioning. In our setting, unlike that of Ercal et al., the final partition during each recursive call had to be into two exactly equal parts. We accomplished this by tolerating a certain level of imbalance during the execution of the Kernighan-Lin heuristic (difference between the two partitions was allowed to be 6% of their size; this value was chosen by experimentation), and then using a greedy method to restore balance at the end.

Asymptotic running time for RMB is $O(lm \log n)$, where l is the average number of stages in the Kernighan-Lin graph partitioning heuristic, which is typically very small. In our experiments RMB gives better solutions than either greedy heuristic for random graphs, geometric graphs, and cubes with additional edges; however, it does poorly on trees. RMB is also faster than all heuristics but SG and LSC. RMB solutions can be used as initial solutions for local search. Results for RMB + LS are very promising, better and faster than G + LS on most classes of graphs.

2.6. Simulated Annealing. Simulated annealing, developed by Kirkpatrick et al. [29], generalizes local search in several ways. The first difference is that the transformation to be applied to the current solution is chosen at random. The second is that uphill moves, transformations that increase solution cost, are allowed, the probability of an uphill move decreasing as the increase in cost gets larger. While these first two differences could easily be incorporated into a sophisticated local-search strategy, the third feature, that the probability of uphill moves gradually decreases throughout a simulated annealing run, is what distinguishes simulated annealing from most local-search variants. An overview of simulated annealing for hypercube embedding is given in Figure 9. Note that the variable T (for temperature), which decreases throughout execution, governs the probability that an uphill move is chosen.

Our implementation of simulated annealing, based on that of Johnson et al. [27] for

```
procedure Map_Subcube($W, d$) is
    /* $W \subseteq V$, $|W| = n/2^d$, the $d$th bit of $f(v)$
        has not been fixed for any $v \in W$ */

    if $|W| = 1$ then return endif
    let $X_0 = \{v \mid d\text{th bit of } f(v) \text{ is fixed at } 0\}$
    let $X_1 = \{v \mid d\text{th bit of } f(v) \text{ is fixed at } 1\}$
    $(W_0, W_1)$ := a partition of $W$ which minimizes the number
                    of edges between $W_0 \cup X_0$ and $W_1 \cup X_1$ subject
                    to the constraint that $|W_0| = |W_1|$
    for $v \in W_0$ do fix $d$th bit of $f(v)$ at 0 end do
    for $v \in W_1$ do fix $d$th bit of $f(v)$ at 1 end do
    Map_Subcube($W_0, d + 1$)
    Map_Subcube($W_1, d + 1$)
    /* at this point the $d$th bit of $f(v)$ is fixed for all $v \in W$ */
end Map_Subcube


procedure RMB($G = (V, E)$) is
    initialize all bits of $f$ so they're not fixed
    Map_Subcube($V, 0$)
end RMB
```

FIG. 8. *Recursive mincut bipartitioning heuristic*

```
$f$ := an initial mapping (usually random)
$T$ := start temperature
repeat
    for some number of iterations do
        choose a random swap $v, w$
        $\Delta := cost(f \oplus \{v, w\}) - cost(f)$
        if $\Delta \leq 0$ then $f := f \oplus \{v, w\}$
         else $f := f \oplus \{v, w\}$ with probability $e^{-\Delta/T}$
    end do
    $T := rT$        /* reduce temperature by temp factor $r$ */
until "frozen"
```

FIG. 9. *Simulated annealing for hypercube embedding*

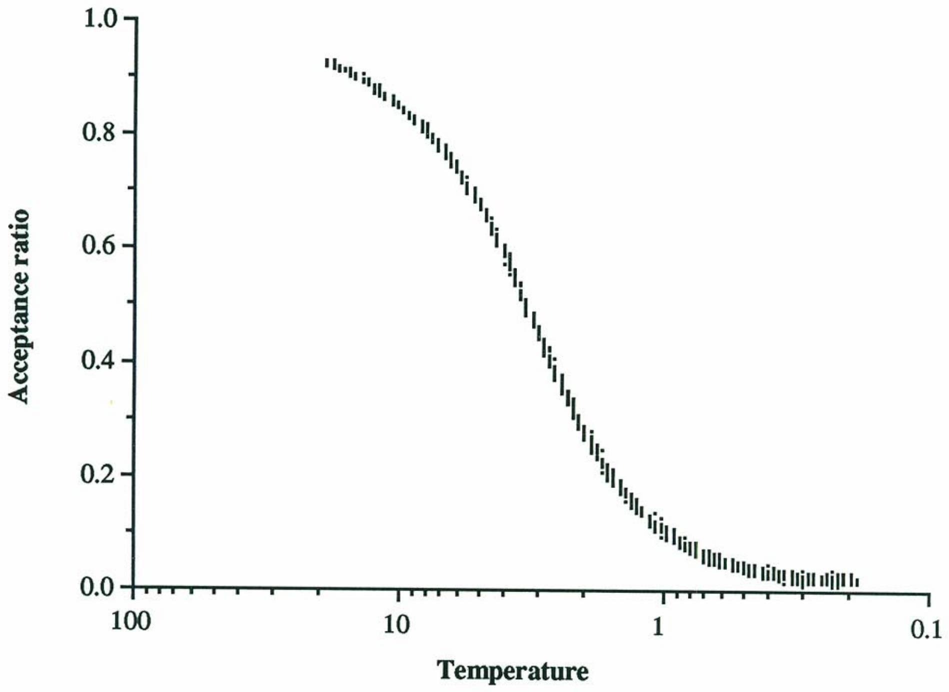
graph partitioning, requires the user to adjust several parameters that affect running time and solution quality. Two of these, *start temperature* and *minpercent*, govern the initial and terminating conditions of a simulated annealing run. At each temperature, the algorithm calculates an *acceptance ratio*, the ratio of swaps actually performed to those considered. The "frozen" condition occurs when no improvement in solution cost has been observed for 5 subsequent temperatures and the acceptance ratio is below *minpercent*. Start temperature is often chosen by doing a trial run to see what temperature gives the desired initial acceptance ratio (usually about 40%). As shown in Figure 10, both solution cost and acceptance ratio decrease as temperature is gradually decreased. The points along the curve represent 20 runs for a random geometric graph using a high start temperature and a low value of *minpercent*. If start temperature is set too high, a lot of time is wasted generating new solutions that are essentially random (the initial flat part of the curve in Figure 10(a)). If it is too low, the advantages of simulated annealing over local search are lost. In the extreme case, when start temperature is close to 0, simulated annealing becomes effectively a local search with random descent and a large number of flat moves. We found that for either random graphs or geometric graphs, increasing the start temperature above 2 does not have a significant effect on solution quality. Data for different start temperatures based on 5 runs on each of the same 10 geometric graphs is shown in Figure 11. We chose 2% as the value for *minpercent*— setting it below that value had almost no effect on solution quality.

More critical to both runtime and solution quality were the choices of *sizefactor*, which governs the number of iterations at each temperature, and *tempfactor*, which governs the rate at which temperature decreases and therefore the expected number of temperatures. The number of iterations at each temperature is a constant, namely *sizefactor*, times the size of the neighborhood (number of possible swaps from each configuration).

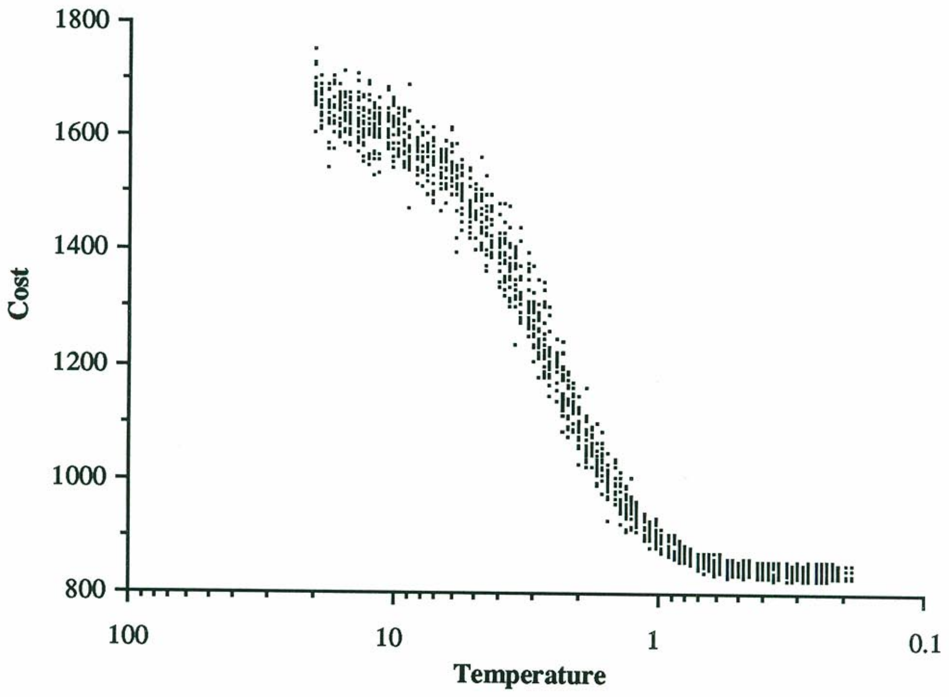
We implemented a version that considers all possible swaps and a version that considers only cube-neighbor swaps. We present results only for the cube-neighbors implementation, called SAC. Experiments comparing the two versions indicate that simulated annealing is powerful enough to overcome the limitations of cube-neighbor swapping. If cost is ignored, an arbitrary swap can be emulated by $O(\log n)$ cube-neighbor swaps. At high temperatures, cost of the intermediate swaps is not a significant factor for simulated annealing, so the power of an all-swaps neighborhood is effectively achieved. Our experiments show that for random graphs, the cube-neighbors approach takes less time to achieve the same solution quality as the all-swaps approach.

The expected asymptotic running time of SAC is $O(lm \log n)$, where l is the expected number of temperatures (for all-swaps simulated annealing, asymptotic time is $O(lmn)$). No data structures were used; the reduction in cost of a swap v, w was calculated simply by looking at the effect on all vertices in $A(v) \cup A(w)$. Maintenance of γ, as used in LSC, but without bucket lists, might improve running time, but only by a small constant factor (runtime per accepted move would increase while runtime for non-accepted moves would decrease).

Figures 12 and 13 illustrate the tradeoff between running time and solution quality



(a)



(b)

FIG. 10. *Progress of simulated annealing*

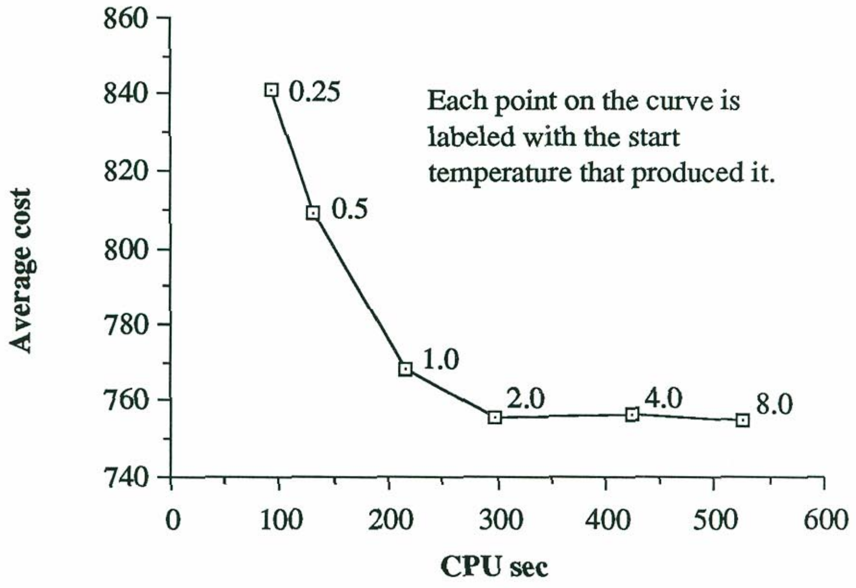


FIG. 11. *Effect of using different starting temperatures*

for varying size and temp factors, respectively. The data are based on 5 runs on 10 geometric graphs for each setting (dimension 7). Other parameters were set at their standard values (start temperature = 2, *minpercent* = 2). The values chosen for the remainder of our experiments (*sizefactor* 16 and *tempfactor* 0.95) were a compromise (see Johnson et al. [27] for discussion of this choice in the context of graph partitioning). After extensive testing on various different types of graphs using a variety of combinations of parameter settings, we concluded that our choice needed to be made somewhat arbitrarily. We were almost always able to improve solution quality significantly by increasing *sizefactor* or *tempfactor* (the only notable exception was when our communication graph was a cube and SAC obtained perfect mappings almost all the time with reasonable settings); running time increased even more significantly when we did this. Without any guidelines on how to judge the tradeoff between time and solution quality we had no basis for saying that any parameter settings were better than any others. To be fair, we included data for two different types of simulated annealing runs on each class of graphs, one with our standard parameters, the other with parameters adjusted so that SAC either had better solution quality than all other heuristics (regardless of time), or competitive solution quality with reduced running time.

The overall idea is that the more time spent annealing, the better the expected solution quality. At one extreme, simulated annealing either degenerates to a variant of local search (start temperature = 0 or a very small *tempfactor*) or limited random probing around an initial solution (small *sizefactor*). At the other extreme, it becomes an exponential algorithm producing nearly optimal solutions with high probability (*note*: this claim has not been verified for hypercube embedding).

Following a recommendation from [27], we improved running time by storing values

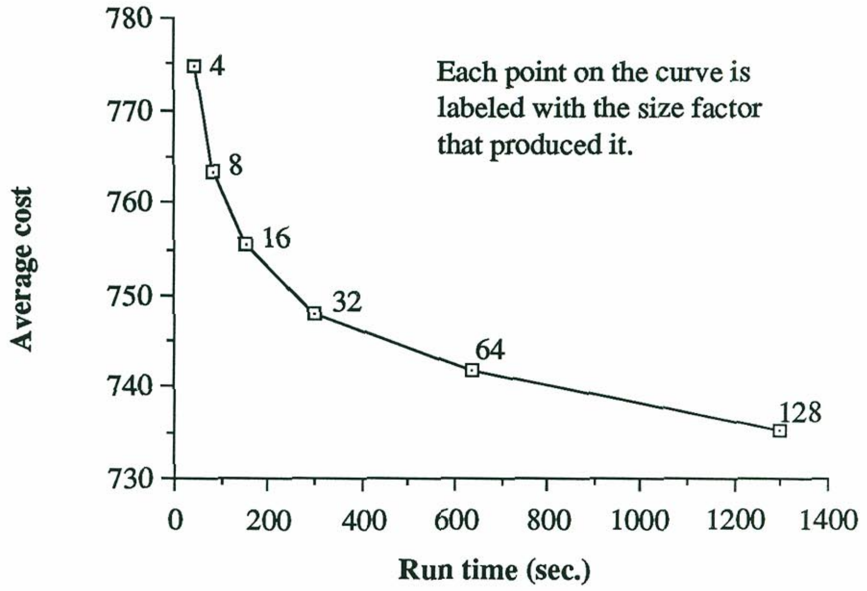


FIG. 12. *Tradeoff between runtime and solution quality for different size factors*

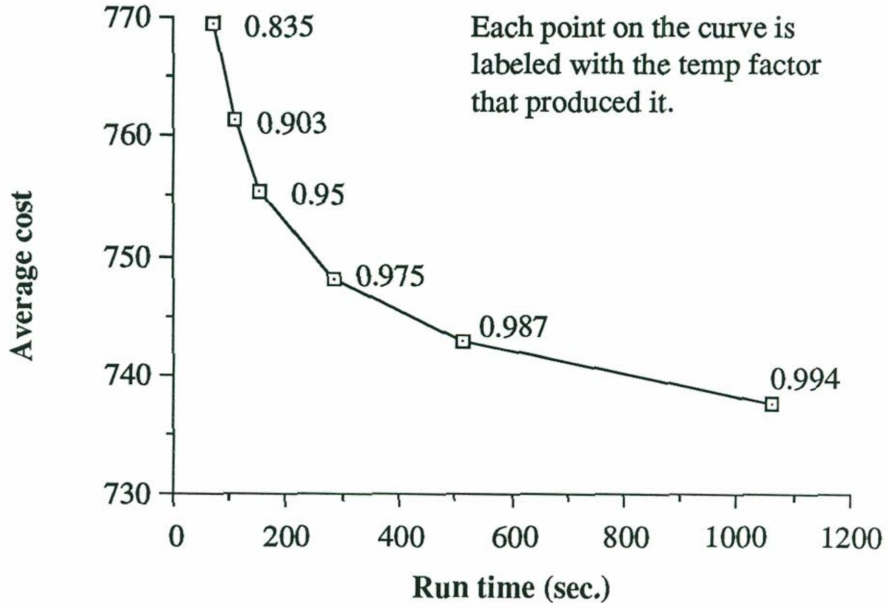


FIG. 13. *Tradeoff between runtime and solution quality for different temp factors*

of e^{-x} in a table for values of x in the range where the probability of an uphill move is significantly different from 0 and significantly different from 1 ($e^{-x} \approx 0.995$ when $x = 1/200$ and ≈ 0.0067 when $x = 5$). Direct calculation of e^{-x} was replaced by the following:

$$\textbf{if } x < -5 \textbf{ then } 0 \textbf{ else } \text{get } e^{\lfloor 200x \rfloor/200} \text{ from table,}$$

that is, e^{-x} was approximated by a table entry for x truncated to the nearest $1/200$ or 0 if $x < -5$. Test runs where we did not approximate the exponent took almost three times as long as identical runs where we did. No improvement in solution quality was observed when the exponent was calculated exactly.

Two final notes on our simulated annealing implementation: Our calculation of the acceptance ratio ignored flat moves. We found that, without this modification, the program sometimes failed to terminate because the presence of large numbers of flat moves prevented it from reaching a low enough acceptance ratio for the frozen criterion. An alternative would have been to choose a higher value for *minpercent*, but this would have unnecessarily penalized the runs that did not have many flat moves. Finally, the solution values we report for simulated annealing were the best values encountered during the run (not necessarily the final solution costs). At the frozen point, the difference between the best value and the final value was usually negligible if there was a difference at all. The primary effect of keeping track of the best solution found so far was to make our results insensitive to the choice of *minpercent*.

3. Experimental Methodology. Here we comment on the two most important aspects of our experimental methodology, our choice of test graphs and the introduction of randomization into all heuristics.

3.1. Types of Communication Graphs. In this paper, four kinds of communication graphs are generated to compare the solution quality of various heuristics. These are random graphs, random geometric graphs, cubes, and trees. The evaluated graphs have exactly the same number of vertices as the hypercubes to which they are mapped. One would expect such graphs to be harder to embed efficiently than graphs with fewer vertices than their target cubes. Our experiments thus stress the algorithms and make comparisons easier.

A random graph $G_{n,p}$ is a graph with n vertices, where each pair of vertices constitutes an edge with probability p. We set p so that the expected number of edges in the random graph is $(n \log n)/2$ (the expected degree of each vertex is $\log n$ and the p is $\log n/(n-1)$.)

The second class of graphs is the random "geometric" graph which is a variant of the one described in [27]. A random geometric graph $U_{n,d}$ has n vertices. The edges of U are generated as follows. First, randomly generate n pairs of numbers uniformly from the interval [0,1), and view each pair as a point in the unit square. These points represent vertices of U; there is an edge between a pair of vertices if and only if their distance is d or less, i.e., if the points are (x_1, y_1) and (x_2, y_2), $|x_1 - x_2| \leq d$ and $|y_1 - y_2| \leq d$. (*Note:* the "infinity norm" was used to compute distances; this simplified the algorithm and did not significantly alter the characteristics of the graphs.) Again, we wanted to make the expected total number of edges in a geometric graph $(n \log n)/2$. Thus the expected degree is $\log n$, the probability of an edge is $\frac{\log n}{n-1}$, and d is $\frac{1}{2}\sqrt{\frac{\log n}{n-1}}$.

Random permutations of hypercubes are often used as communication graphs to evaluate local-search heuristics [2, 10]. Since cubes can be embedded in themselves exactly, they can be fairly good indications of how close heuristic solutions are to optimal solutions. In order to make the problem more difficult, we also considered cubes with a number of edges randomly added (or deleted). The number of edges to be added in our experiments was chosen somewhat arbitrarily to be the dimension of the cube. Two types of cubes with edges deleted were tested: one with few deletions (number of deletions = dimension) and the other with more deletions (number of deletions = $(n \log n)/16$). Results for few deletions were very similar to those for exact cubes while more deletions produced results more like those for random graphs.

Random trees are also used as communication graphs. Many parallel algorithms are tree structured, since trees are natural to data structures and divide-and-conquer algorithms. Also, they are the best candidates for experiments on very sparse graphs. An n-vertex random tree T_n was generated by the algorithm in Figure 14.

```
$S := \{v\}$, where $v$ is a random vertex in $T$
    /* $T$ is the vertex set of $T_n$ */
**repeat**
    $v :=$ a randomly chosen vertex from $S$
    $w :=$ a randomly chosen vertex from $T - S$
    Add the edge $\{v, w\}$ to $T_n$
    $S := S \cup \{w\}$
**until** $S = T$
```

FIG. 14. *Algorithm for generating random trees*

3.2. Randomization. Since simulated annealing is a randomized algorithm, obtaining different results for different runs on identical data, we decided to randomize all the heuristics for fair comparison. The iterative improvement heuristics are naturally randomized by the choice of a random initial solution. They can be further randomized by choosing randomly among equally desirable alternatives. For example, in the case of steepest descent there are often several swaps giving the best cost decrease. This second type of randomization appears to help, at the expense of a slight increase in runtime. The iterative improvement heuristics other than SAC used in our experiments only did randomization with the starting solution and in the choice of a flat move; downhill moves were chosen arbitrarily rather than randomly.

For RMB, randomness is present in the choice of a random starting partition during each recursive call. Greedy heuristics were the most difficult to randomize. For Chen's greedy heuristic we randomized the sequence for processing hypercube vertices during updates of *gain* (the inner **for** loop in Figure 3). If a communication-graph vertex had the same gain for two or more different hypercube vertices ($gain(v, h_1) = gain(v, h_2)$, for example), the corresponding entries appear on the bucket list in random order. For the simple greedy heuristic we simply permuted the vertices randomly before initially inserting them into the bucket list. This random sequence influenced the order among equal alternatives later in the bucket list and gave rise to surprisingly large variances in the results obtained.

To compare the average performance of heuristics for each kind of graph, we made 5 runs on each of 10 graphs. Since the heuristics are all randomized, one run on one graph may not be conclusive. In our experiments, all heuristics were run on the same set of test graphs. Thus the average solution cost of each heuristic reflects its relative solution quality. Also, the iterative improvement heuristics were given the same set of initial solutions for each run on each graph, so that they began at the same starting point. In cases where G, SG, or RMB was used as a front end to LS, LSC, or KL, the results generated by the former were fed directly to the latter. Thus again, LS, LSC, and KL were started with the same set of initial solutions and always generated results better than their front ends (e.g., the cost obtained by G+LS is always better then pure G).

We accomplished the controlled randomization described above by providing three distinct random number streams: one for communication-graph generators, one for heuristics used as front ends (i.e., random initial solution generator, G, SG, and RMB), and one for iterative improvement heuristics. Thus our results for combined heuristics were not influenced by the effects altering the random number stream of each individual heuristic.

4. Results. As already explained in Section 3, random graphs, geometric graphs, trees, and hypercubes are used as communication graphs to evaluate the heuristics. Observe that the structure of these four kinds of graphs advances gradually from random to exact hypercubes. Random graphs have no obvious relation to hypercubes. Geometric graphs are easier to partition, and thus structurally closer to hypercubes. Trees are always subgraphs of hypercubes if we allow arbitrarily large dimensions, but the embedding of a tree into a *fixed-size* hypercube can have large average dilation (for example, consider embedding a star with 2^k vertices into H_k). Communication graphs that are hypercubes with a few added edges (or deleted edges) are always supergraphs (or subgraphs) of hypercubes. Consequently, the results illustrate variations across a continuum of structures. In particular, simulated annealing performs better as the communication graph becomes more random, and greedy strategies perform better as the communication graph approaches a hypercube.

We report experimental results for a total of twelve heuristics for hypercube embedding (those described in Section 2 used singly or in combinations):

- greedy (G);
- simple greedy (SG);
- all-swaps, steepest-descent local search (LS);

- cube-neighbors, steepest-descent local search (LSC);
- Kernighan-Lin (KL);
- recursive mincut bipartitioning (RMB);
- cube-neighbors simulated annealing (SAC);
- greedy initial solution followed by LS (G+LS);
- G+LSC;
- G+KL;
- RMB+LS; and
- SG+LSC.

The reader should bear in mind that our results should be viewed as general indications of what types of techniques work well on what types of graphs rather than specific endorsements of one heuristic over another.

The simple greedy algorithm was easily the fastest heuristic we tested, but produced by far the poorest mappings. Apart from SG, LSC was fastest, but also gave the next highest cost solutions. SAC was slowest but produced the best solutions. In general, SAC's solutions improved on LSC's by about as much as LSC's improved on SG's.

Ignoring SG for the moment, the gap in solution cost (on random graphs) between the fastest algorithm, LSC, and the slowest, SAC, was never more than about 10%. This was also true for selected random graphs with 64, 256 and 512 vertices mapped onto hypercubes of dimension 6, 8 and 9 (see Figure 15; data is based on 20 runs on one graph). This 10% improvement in solution cost comes at the expense of a factor of about 300 in the running time, not an especially good bargain unless the parallel program in question is to be executed over a long period of time.

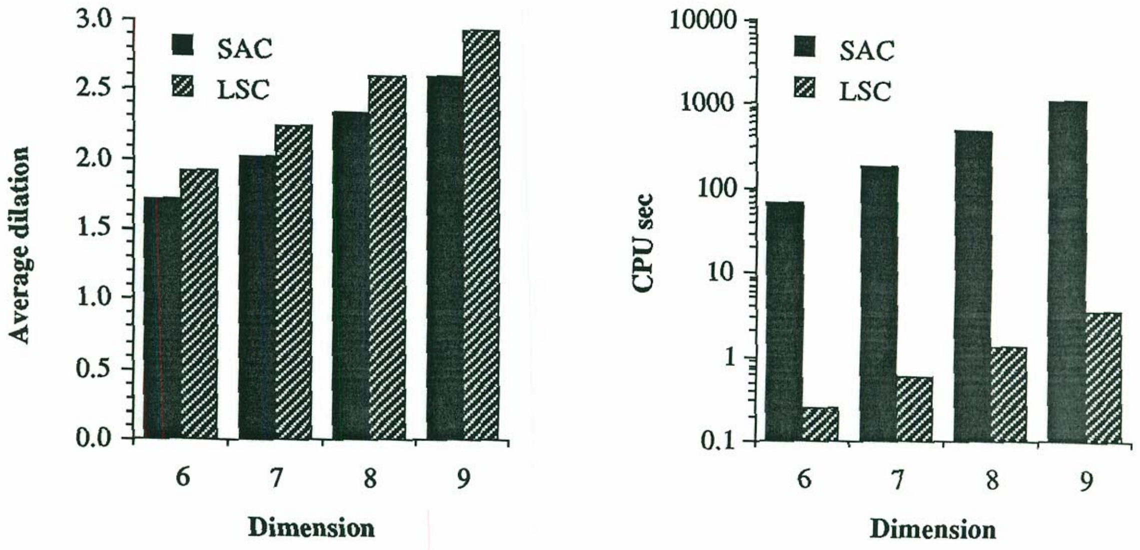


FIG. 15. *Comparison of SAC with LSC*

This observation also suggests that, rather than improving the sophistication of existing heuristics, at the expense of increasing running time, more effort should go into the development of fast and simple heuristics whose solutions fall within this 10%

threshold. Our LSC heuristic is a step in that direction. We also expect that a slightly more sophisticated variant of SG will meet this goal.

Although the differences in solution quality in most experiments are not great, it is worth noting that (a) there are situations when nearly optimal mappings are worth additional computation time, (b) other applications of hypercube embedding, such as coding theory, may require high quality embeddings, (c) low-cost mappings are more important in large cubes, especially when link contention is taken into account, and (d) some of our results show an interesting trend in the efficacy of simulated annealing versus other heuristics on random versus structured problems (similar results are reported for graph partitioning by Johnson et al. [27]).

In general LS, LSC, and KL do better when they have better initial solutions. Thus, in both running time and solution quality, G+LS dominates LS, G+KL dominates KL, SG+LSC dominates LSC, and RMB+LS dominates LS.

At the extremes of solution quality, two questions arise naturally: (a) how much better is SG than random? and (b) how close is SAC to optimal? The experiments demonstrate that SG betters random mappings by a large margin, and that SAC also beats SG by a large margin. It is difficult to say whether SAC is close to optimal for the "average" communication graph. The experiments on cube-structured communication graphs do confirm that the "pushed" SAC (SAC with higher size and/or tempfactor) is capable of obtaining optimal solutions all the time, and that SAC with our standard parameters maps hypercubes to hypercubes with near perfection. However, for random graphs the story may be completely different, making it difficult to draw general conclusions about the optimality of SAC.

The tables within each subsection include data showing solution quality and runtime for the various heuristics. We also show the average and the minimum cost of a randomly generated solution as a basis for comparison. For each heuristic we indicate how its average solution compared with an average random solution (the "% of random" column gives the ratio $\frac{\text{heuristic cost}}{\text{random cost}}$ as a percentage).

4.1. Random graphs. As shown in Table 2, the results for random graphs exhibit a typical tradeoff between running time and solution quality. Simulated annealing is at its best on random graphs. As the table shows, SAC beats all other heuristics except for G+KL and KL in solution quality by a wide margin. Runtime for SAC with standard parameter settings is not competitive, however. We included a relaxed version of SAC in the table to show the that reasonable solution quality can be achieved by SAC without outrageous runtime.

At one end of the runtime/solution-quality spectrum, the most interesting story is the competition between SAC and G+KL. For either, solution cost can be reduced by adjusting parameters (*sizefactor* and *tempfactor* in the case of SAC, uphill moves in the case of G+KL). We were unable to find a leveling-off point for either heuristic, a point at which the improvement in solution quality achieved by increased runtime becomes negligible. It appears that G+KL has lower runtime than SAC for the same solution quality. This observation must be tempered by two cautionary remarks. First, SAC has better asymptotic runtime and may in fact do better for larger cubes. Second, SAC

TABLE 2

Results for mapping random graphs

Heuristic	Min	Avg. Cost	Avg. Dila.	% of random	CPU sec.
SAC*	844	916.3	2.042	58.1 %	175.17
G+KL*	855	923.7	2.059	58.5 %	93.79
KL	859	925.7	2.064	58.7 %	98.55
RMB+LS*	865	942.3	2.101	59.7 %	11.28
SAC(relaxed)[a]	874	948.2	2.114	60.1 %	25.29
G+LS	869	954.0	2.127	60.5 %	12.48
LS	888	963.5	2.148	61.1 %	15.85
G+LSC*	901	982.5	2.190	62.3 %	1.08
RMB*	916	1003.9	2.238	63.6 %	0.94
SG+LSC*	922	1009.1	2.249	63.9 %	0.45
LSC	930	1013.6	2.259	64.2 %	0.49
G	943	1024.7	2.284	64.9 %	0.86
SG*	1189	1286.1	2.867	81.5 %	0.03
Random	1472	1578.0	3.518	100.0 %	—

[a] *sizefactor* = 2

* Indicates a "competitive" heuristic, one with a lower runtime than all heuristics that achieved a lower average cost.

can be implemented in linear space, while both G and KL require space proportional to n^2. Space becomes a significant limiting factor for n^2-space heuristics on cubes of dimension 10 or more.

In the middle of the range, the best competitors are various local-search variants with better than random starting solutions. RMB appears to be an excellent choice for starting solutions; even RMB by itself is competitive, especially when its asymptotic runtime is taken into account (asymptotic runtime for RMB is much better than G+LSC and about the same as SG+LSC). Some midrange entries that do not appear in the table are RMB+LS with no flat moves and a cube-neighbors Kernighan-Lin implementation with a greedy initial solution (no uphill moves). Both fall into the gap between LS and G+LSC in terms of solution quality and have runtimes significantly better than that of RMB+LS with flat moves (but not quite as good as G+LSC).

If runtime is the major consideration, SG is the best among the heuristics we tested. SG+LSC is a good choice with fast runtime and reasonable solution quality. The gap between G and SG can be filled by other greedy heuristics, for example, a more sophisticated version of SG.

Since the statistics in the table come from 10 different random graphs, the minimum cost reported is not likely to be robust — all the minimum cost data come from the same graph, so this statistic is really based on 5 runs rather than 50. Also, the gap between minimum and average cost is not a true indicator of variance. To make up for these shortcomings Figure 16 shows histograms based on 100 runs of each of four heuristics (SAC, SG, G+KL, and SG+LSC). All runs are on the same random graph. Note that the heuristics with better solution quality have less variance. This is a general trend that holds for all of the heuristics we tested.

An interesting question to ask is: what is the expected minimum cost solution for each heuristic if we do as many runs of it as can be accommodated within a fixed amount of time? For example, will 100 runs of RMB+LS give a better minimum solution than 6 runs of SAC (the fixed amount of time is roughly 1000 CPU seconds)? On random graphs this sort of competition appears to favor simulated annealing (Johnson et al. [27] did a more extensive study for the graph partitioning problem, where simulated annealing was also the winner on random graphs).

4.2. Geometric graphs. Table 3 reports results for geometric graphs. The general trends are similar to those of random graphs but there are some important differences. The most important difference is that SAC is no longer a clear leader. Geometric graphs have enough structure that other heuristics can exploit while simulated annealing does a lot of random probing. As the first line of the table indicates, we can always push SAC to the point where its solution cost significantly beats other heuristics, but with significant increases in runtime.

TABLE 3
Results for mapping random geometric graphs

Heuristic	Min	Avg. Cost	Avg. Dila.	% of random	CPU sec.
SAC(pushed)[a]	694	742.8	1.691	48.0 %	565.85
G+KL*	694	743.6	1.693	48.1 %	84.55
RMB+LS*	698	751.0	1.709	48.5 %	11.90
KL	707	754.9	1.718	48.8 %	101.50
SAC	709	755.3	1.719	48.8 %	169.53
G+LS	705	757.8	1.725	49.0 %	12.34
LS	727	781.9	1.780	50.5 %	18.52
G+LSC*	720	783.2	1.783	50.6 %	0.93
RMB	733	791.3	1.801	51.1 %	0.94
SG+LSC*	748	797.8	1.816	51.6 %	0.30
G	737	804.8	1.832	52.0 %	0.76
LSC	802	864.6	1.968	55.9 %	0.44
SG*	846	916.3	2.086	59.2 %	0.03
Random	1451	1547.5	3.523	100.0 %	—

[a] *sizefactor* = 32, *tempfactor* = 0.975

Another significant difference is the improved position of the greedy heuristics. Greedy heuristics are at their best on highly structured graphs, and we begin to see this phenomenon even in the relatively unstructured random geometric graphs.

As with random graphs, we include some histograms in Figure 17. Except for the

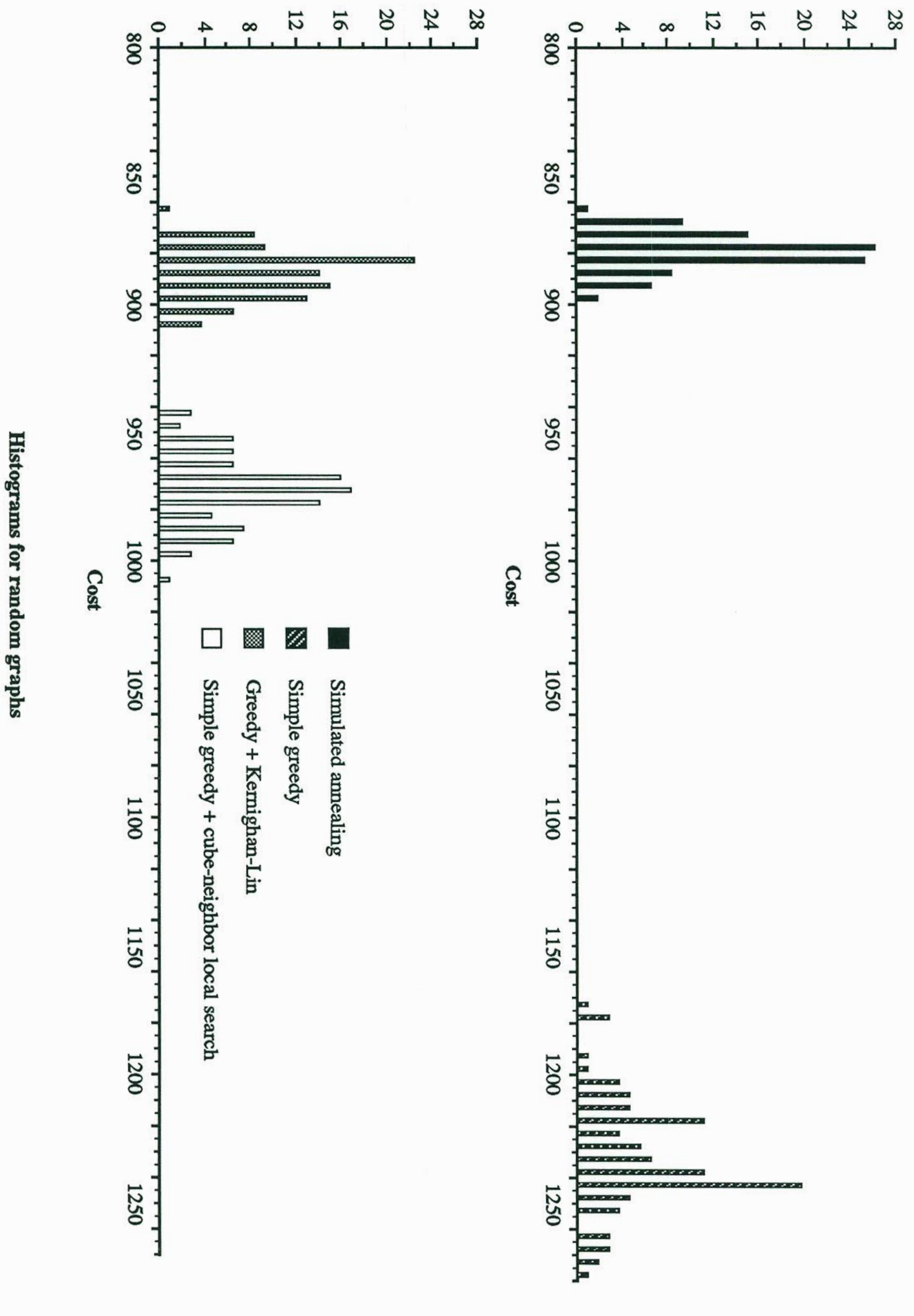


FIG. 16. *Histograms of heuristics on random graphs*

slight overlap between G+KL and SG+LSC and the lower cost of SG by itself, the picture is similar to that of random graphs.

Note also that for most heuristics, the solution cost for geometric graphs is nearly 20% less than for random graphs even though the average number of edges in these graphs is roughly the same. This is not surprising, since geometric graphs are structurally closer to hypercubes than random graphs are.

4.3. Trees. For trees, as shown in Table 4, the absolute cost difference between heuristics is much smaller than for either random graphs or geometric graphs, but the relative cost difference is actually larger. The best average dilation obtained by any run of any heuristic was 1.06 which is quite close to 1, the theoretical minimum. The worst average dilation obtained by SG+LSC was 1.35.

TABLE 4
Results for mapping random trees

Heuristic	Min	Avg. Cost	Avg. Dila.	% of random	CPU sec.
SAC(pushed)*[a]	136	140.7	1.108	31.5 %	797.17
G+KL*	140	144.2	1.135	32.3 %	21.34
SAC	140	144.5	1.138	32.3 %	203.76
G+LS*	137	144.6	1.139	32.4 %	3.58
KL	142	148.2	1.167	33.2 %	36.24
RMB+LS	141	150.3	1.183	33.6 %	4.73
G+LSC*	144	150.6	1.186	33.7 %	0.56
LS	145	156.0	1.228	34.9 %	7.00
G	148	157.7	1.241	35.3 %	0.44
SG+LSC*	156	165.0	1.299	36.9 %	0.21
RMB	156	169.3	1.333	37.9 %	0.76
LSC	168	175.4	1.381	39.2 %	0.33
SG*	200	214.7	1.691	48.0 %	0.02
Random	408	446.9	3.519	100.0 %	—

[a] *sizefactor* = 64

Since trees are more structured than geometric graphs, it is not surprising that Chen's greedy heuristic, G, is a strong competitor. SG, however, is not sophisticated enough to take full advantage of the structure of trees. Choosing the next vertex to embed based only on how many neighbors have already been embedded is not a good strategy for trees, which have low connectivity. G is also better than RMB on trees, both by itself and as a front end to iterative improvement heuristics. For trees there are many close to optimal partitionings, some resulting in good embeddings, others not. Partitioning by itself is not a good indicator of average dilation for trees. The greedy heuristic, on the other hand, tries to map large connected components of each tree perfectly, and ends up having to sacrifice (i.e. assign large dilation to) only a small number of edges in the process.

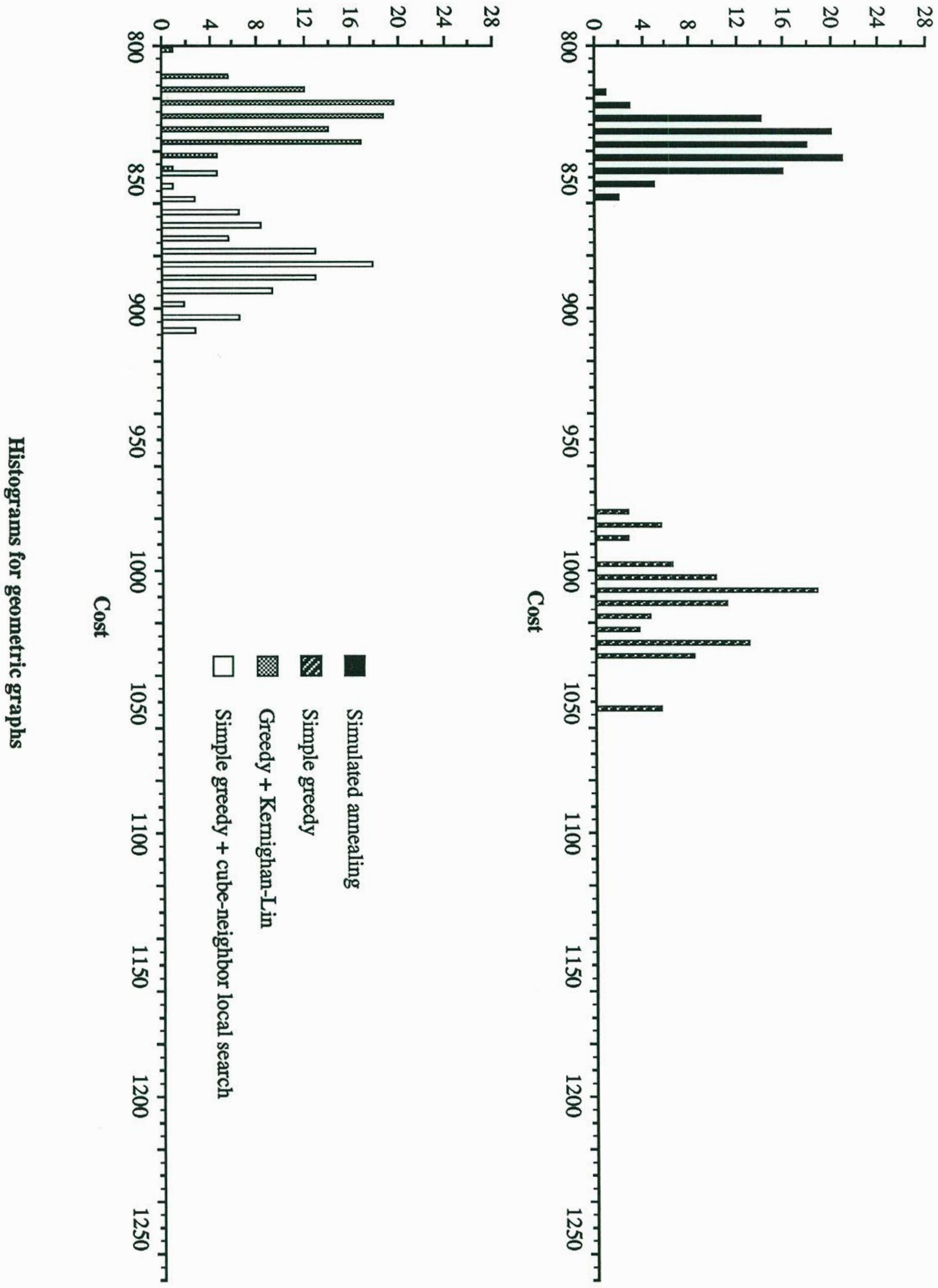


FIG. 17. *Histograms of heuristics on geometric graphs*

Another significant phenomenon for trees is the importance of flat moves in iterative improvement heuristics. Simulated annealing takes a lot of time, more than for random graphs, because many of the moves that are accepted lead to no improvement. On the other hand, G+LS with 640 flat moves (not shown in the table) achieves an average solution cost of 139.5 with an average runtime of only 20.76. The average dilations achieved by this combination are evidently close to optimum.

4.4. Cubes. In this section we report results for cubes and graphs that are very close to cubes. There exist efficient algorithms for mapping cubes to themselves exactly [6], so results for heuristics mapping exact cubes are of questionable practical importance (we are not aware of any exact algorithms for mapping cubes with a small number of edges added or deleted, however). The results for cubes and near-cubes do illustrate some interesting points. Cubes are the most structured communication graphs possible for this problem. Hence the greedy heuristics and RMB tend to outperform the iterative-improvement heuristics. Cubes and near-cubes are also the only communication graphs for which we know the cost of the optimal solutions, so we can, in this idealized setting, judge the solutions obtained in relation to known optimum solutions. Finally, it can be argued that in order for a heuristic to be of practical interest, it should do a reasonable job on graphs that have perfect mappings. Cubes and cubes with edges deleted are a good way to test this quality.

Table 5 gives our data for mapping randomly permuted cubes. Chen's greedy heuristic always obtains the optimal solution in this case (as does the heuristic of Lee and Aggarwal). Obviously this eliminates the need to include data for G+LS or G+LSC. If SAC is pushed slightly it also obtains optimal solutions consistently (an observation also made for the simulated annealing implementation of Ramanujam et al. [36]). Most of the other heuristics manage to obtain at least one optimal solution.

TABLE 5
Results for mapping randomly permuted cubes

Heuristic	Min	Max	Avg. Cost	Avg. Dila.	% of random	CPU sec.
G*	448	448	448.0	1.000	28.3 %	0.76
RMB	448	448	448.0	1.000	28.3 %	0.81
SAC(pushed)[a]	448	448	448.0	1.000	28.3 %	274.11
SAC	448	576	460.2	1.027	29.0 %	69.59
KL	448	816	541.7	1.209	34.2 %	147.09
SG+LSC	448	764	636.2	1.420	40.1 %	0.08
SG*	448	774	639.3	1.427	40.3 %	0.03
LS	710	890	832.0	1.857	52.5 %	24.31
LSC	876	962	920.1	2.054	58.0 %	0.66
Random	1500	1652	1585.7	3.539	100.0 %	—

[a] *sizefactor* = 64, *tempfactor* = 0.975

The only heuristics not obtaining at least one optimal solution in 50 runs are LS and

LSC. In fact LS and LSC don't do much to improve the quality of their initial solutions unless these are random (compare results for SG+LSC with those for SG). Figure 18 illustrates a serious shortcoming of LSC when mapping cubes. If two opposite corners on the same face of a 3-cube are swapped, the resulting mapping is a local optimum, even if we allow flat moves. No swap of cube neighbors can be done without strictly increasing cost. This trap does not exist for LS which allows swaps between two arbitrary vertices. Figure 19 illustrates a local optimum on a 4-cube for LS. One of the two main subcubes is rotated with respect to the other. Every individual swap will strictly increase cost (for example, in order to align 1111 with 0111 one of the two has to be moved away from its 3 neighbors in the subcube). The number of such local optima (and their cost relative to the optimum cost) increases dramatically as cube dimension increases.

There are several ways to avoid getting stuck at local optima. Flat moves, which appear to be promising for more random graphs, are useless here. Simulated annealing avoids the traps by doing random uphill moves. KL does so by doing both random and systematic uphill swaps.

The results for cubes with 7 edges deleted, reported in Table 6, are similar to those for cubes. With more edges deleted (56 was chosen arbitrarily), the resulting graphs are considerably more random and the sequence of contenders looks more like that for geometric graphs — results are shown in Table 7. The main differences between the results for cubes minus 56 edges and those for geometric graphs are that (a) RMB does significantly better for the cube subgraphs, and (b) the local search variants by themselves do significantly worse for the cube subgraphs. The influence of the cube structure is a factor, but not enough of one to make Chen's greedy heuristic competitive. This is because Chen's heuristic (or any other greedy heuristic) examines local structure, which for cubes missing many edges looks random. RMB, with its top-down recursive approach, takes advantage of global structure, which appears to be the overriding factor here. Note that the 392 minimum achieved by most of the heuristics is the optimum solution.

For cubes with 7 edges added the results are reported in Table 8. Here the clear-cut champion is RMB. Local search does badly for reasons already suggested. Simulated annealing gets good solutions only at the expense of long runtime (pushing SAC beyond the factors shown in the table merely increased runtime but did not improve average solution cost). There are enough extra edges to fool Chen's greedy heuristic, preventing optimum solutions. We suspect that the 470 minimum attained by about half of the heuristics is an optimum solution.

5. Conclusions.

A variety of conclusions can be drawn from the results we report. In each of the following paragraphs we highlight one major observation we were able to make.

The heuristics we studied exhibit a wide range of options along a continuum. There appears to be a clear tradeoff between solution quality and runtime. Often it is possible to obtain several favorable points along the continuum by modifying a single heuristic (adding flat moves or random jumps to local search, varying parameters for simulated annealing, adding more sophisticated choice mechanisms to greedy heuristics).

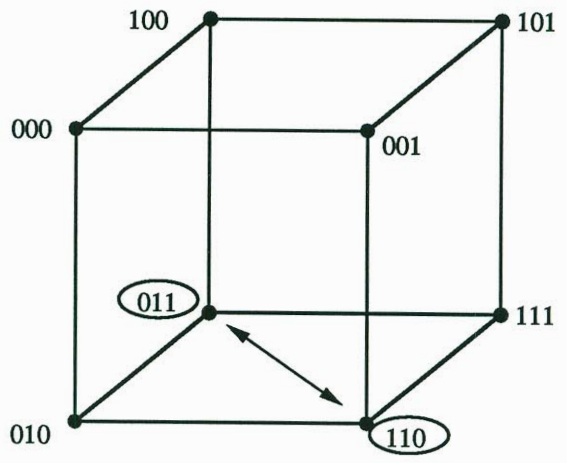


FIG. 18. *Local optimum for the LSC heuristic*

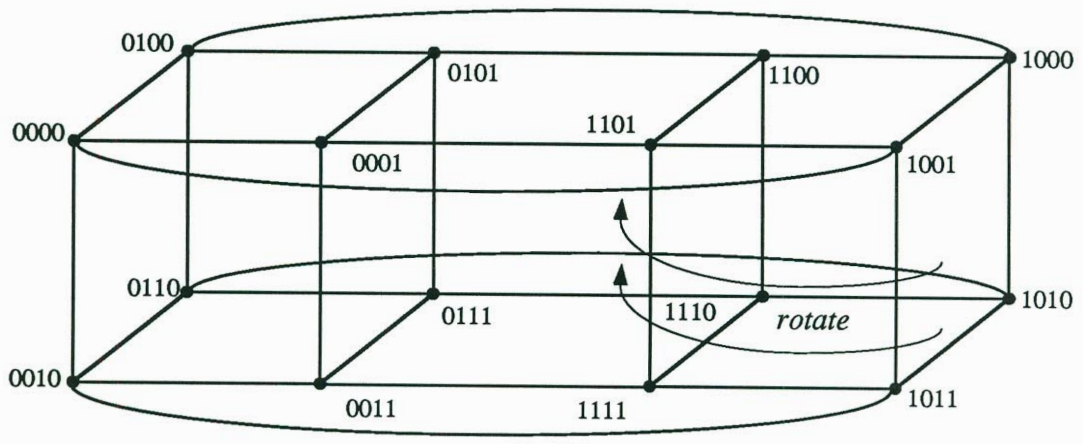


FIG. 19. *Local optimum for the LS heuristic*

TABLE 6

Results for mapping cubes with 7 edges deleted

Heuristic	Min	Avg. Cost	Avg. Dila.	% of random	CPU sec.
G*	441	441.0	1.000	28.2 %	0.76
SAC(pushed)[a]	441	441.0	1.000	28.2 %	538.60
RMB	441	444.9	1.009	28.5 %	0.80
RMB+LS	441	444.9	1.009	28.5 %	1.29
SAC	441	457.2	1.037	29.3 %	75.20
KL	441	525.1	1.191	33.6 %	162.57
SG+LSC*	587	778.8	1.766	49.8 %	0.15
LS	743	831.1	1.884	53.2 %	21.44
SG*	717	870.1	1.973	55.7 %	0.03
LSC	847	912.9	2.070	58.4 %	0.62
Random	1508	1563.0	3.544	100.0 %	—

[a] *sizefactor* = 128, *tempfactor* = 0.975

TABLE 7

Results for mapping cubes with 56 edges deleted

Heuristic	Min	Avg. Cost	Avg. Dila.	% of random	CPU sec.
SAC(pushed)[a]	392	392.0	1.000	28.3 %	775.38
G+KL*	392	394.2	1.006	28.5 %	32.73
RMB+LS*	392	415.5	1.060	30.0 %	1.54
SAC	392	426.5	1.088	30.8 %	102.10
RMB*	392	436.2	1.113	31.5 %	0.81
G+LS	392	437.2	1.115	31.6 %	2.52
KL	392	448.9	1.145	32.4 %	135.74
G+LSC	392	449.6	1.147	32.5 %	0.79
G	392	458.4	1.169	33.1 %	0.71
LS	662	734.5	1.874	53.1 %	16.11
SG+LSC*	657	747.2	1.906	54.0 %	0.30
LSC	766	810.1	2.066	58.5 %	0.44
SG*	861	931.1	2.375	67.3 %	0.03
Random	1336	1384.5	3.532	100.0 %	—

[a] *sizefactor* = 128, *tempfactor* = 0.975

TABLE 8

Results for mapping cubes with 7 additional edges

Heuristic	Min	Avg. Cost	Avg. Dila.	% of random	CPU sec.
SAC(pushed)*[a]	470	473.9	1.042	29.4 %	272.06
RMB*	470	475.1	1.044	29.5 %	0.82
RMB+LS	470	475.1	1.044	29.5 %	1.33
G+KL	470	486.5	1.069	30.2 %	47.02
SAC	470	489.5	1.076	30.4 %	72.39
KL	470	565.8	1.244	35.1 %	149.69
G+LS	470	571.1	1.255	35.4 %	3.72
G+LSC	470	623.0	1.369	38.6 %	0.90
G	475	659.3	1.449	40.9 %	0.78
SG+LSC*	772	862.1	1.895	53.5 %	0.22
LS	772	865.7	1.903	53.7 %	23.33
LSC	872	948.7	2.085	58.8 %	0.60
SG*	785	962.2	2.115	59.7 %	0.03
Random	1554	1612.2	3.543	100.0 %	—

[a] *sizefactor* = 64, *tempfactor* = 0.975

Iterative improvement heuristics perform better when the starting solution is better than random (greedy or recursive mincut bipartitioning are good methods for generating starting solutions). As the sophistication of the iterative improvement heuristic increases, the difference made by better-than-random starting solutions is less pronounced (for example, solutions obtained by KL and SAC are not significantly improved when greedy initial solutions are used).

The choice of transformations (neighborhood) has an impact on the quality of solutions obtained by iterative improvement heuristics. As is the case with better starting solutions, the effect of choosing a more powerful (i.e., larger) neighborhood diminishes as the sophistication of the underlying heuristic improves. For local search, a strategy that allows all swaps is significantly better than one that only allows swaps between cube neighbors. Unfortunately a larger neighborhood also has the effect of increasing runtime significantly. Hence, with sophisticated heuristics such as simulated annealing or Kernighan-Lin, it may be better to choose a more limited neighborhood.

The structure of the communication graph must be taken into account when choosing a heuristic. For completely random graphs, the best choices tend to be iterative improvement heuristics. For graphs that are cubes or almost cubes, greedy heuristics are the best choice. The advantages of simulated annealing diminish as the communication graphs become more structured. The type of structure may also be a factor. For example, RMB did poorly on trees, which have local structure, but well on cubes with edges deleted, where the structure in relation to exact cubes is more global.

Simulated annealing, because of the wide range of results obtainable by adjusting parameters, is the most versatile heuristic we tested. At the standard parameter settings we chose, simulated annealing obtained consistently good results on all runs for all graphs. Other heuristics had larger variances for runs on a single graph or were ineffective for certain classes of graphs. The two primary disadvantages of simulated annealing are the large runtimes and the effort required to adjust parameters.

6. Further Research. One of our aims is to stimulate further research into the many questions raised by the performance of the heuristics we evaluated. Our initial goal was to implement and test efficient versions of several standard heuristics already reported in the literature. By doing extensive testing on several different kinds of graphs we sought to gain insights about the relative efficacy of different kinds of heuristics for different kinds of communication graphs. We found that, in pitting so many heuristics against each other, a natural tendency to improve the competitiveness of each heuristic emerged. If the performance of one heuristic is enhanced by a particular maneuver, it is natural to consider the effect of a similar enhancement on other heuristics.

Many of the heuristics were sensitive to adjustments of various parameters which affected tradeoff between runtime and solution quality. The most pressing need that we felt in the process of adjusting parameters was for some kind of analytical model of what constitutes a good tradeoff between runtime and solution quality. Is it beneficial to double the runtime in order to decrease the average solution cost by 10 units? Questions such as this can only be answered if more is known about the specific application. Most of our attempts to meld statistics on runtime and solution quality into a single number favored the fastest heuristics (unless we postulate that runtime should be exponentially related to the difference between average solution cost and some estimate of optimal solution cost). Analytical models of the tradeoff and practical justifications for them would be extremely useful in any further research of this kind. Without such models it is difficult to establish fair comparisons among heuristics with adjustable parameters.

Hypercube embedding is an unusually difficult problem, even among NP-hard graph problems. It is NP-hard even for trees. Most successful heuristics reported in the literature have asymptotic running times of $O(n^3)$ or worse. Experimental data, including ours, has been limited to relatively small problems. Lower bounds on the cost of an optimal solution appear to be difficult to obtain. We propose the following items as worthy of future experimental study.

- Development of a reasonable exhaustive search or branch-and-bound strategy so that optimal solutions for random test graphs can be generated.
- More tests on larger problems (dimension 10, for example) to see if the relative standing of the heuristics holds up asymptotically.
- Variations to improve the solution quality of the simple greedy heuristic.
- Extensive testing of variations on the KL heuristic to determine the relative merits of flat moves versus random jumps and all swaps versus cube neighbors.
- More testing of a Kernighan-Lin type heuristic based on cube-neighbor swaps (the success of simulated annealing with cube-neighbor swaps suggests that this combination may also do well; preliminary experiments with a cube-neighbors KL implementation are promising — see [15]; Ercal and Sadayappan [18] also report experiments with a cube-neighbors Kernighan-Lin heuristic, but their

strategy does not consider uphill moves).

- More testing and refinement of the RMB heuristic (with better graph partitioning heuristics).
- More testing to determine the relative merits of flat moves versus Bokhari-style random jumps as enhancements to local search heuristics (random jumps appear to give better solutions based on some preliminary experiments, but more testing needs to be done).
- Implementation and testing of iterative improvement heuristics based on subcube rotations rather than swaps (the example in Figure 19 suggests that subcube rotations might be an effective method for sidestepping local optima; cube-neighbor swaps are rotations of subcubes of dimension 1; a more general strategy that allows rotations of subcubes of dimensions 2 and/or 3 might be worth considering)
- More testing to determine the effect of sparsity on various heuristics (with the exception of trees, our test graphs all had roughly $(n \log n)/2$ edges).

REFERENCES

[1] F. Afrati, C. Papadimitriou, and G. Papageorgiou, *The complexity of cubical graphs*, Information and Control, 66 (1985), pp. 53 – 60.

[2] F. André, J. Pazat, and T. Priol, *Experiments with mapping algorithms on a hypercube*, in Proceedings Fourth Conference on Hypercubes, Concurrent Computers, and Applications, 1989.

[3] F. Berman, *Experience with an automatic solution to the mapping problem*, in The Characteristics of Parallel Algorithms, L. Jamieson, D. Gannon, and R. Douglass, eds., MIT Press, 1987.

[4] F. Berman and L. Snyder, *On mapping parallel algorithms into parallel architectures*, Journal of Parallel and Distributed Computing, 4 (1987), pp. 439 – 458.

[5] S. Bettayeb, Z. Miller, and I. Sudborough, *Embedding grids into hypercubes*, in VLSI Algorithms and Architectures: 3rd Aegean Workshop on Computing, Lecture Notes in Computer Science 319, Springer Verlag, 1988, pp. 201 – 211.

[6] K. Bhat, *On the complexity of testing a graph for N-cube*, Information Processing Letters, 11 (1980), pp. 16 – 19.

[7] S. Bhatt, F. Chung, F. Leighton, and A. Rosenberg, *Efficient embeddings of trees in hypercubes*. Typescript, Department of Computer Science, Yale University, New Haven, CT 06520.

[8] L. Bhuyan and D. P. Agrawal, *Generalized hypercube and hyperbus structures for a computer network*, IEEE Transactions on Computers, C-33 (1984), pp. 323–333.

[9] S. Bokhari, *On the mapping problem*, IEEE Transactions on Computers, C-30 (1981), pp. 207 – 214.

[10] S. Bollinger and S. Midkiff, *Processor and link assignment in multiprocessors using simulated annealing*, in Proceedings International Conference on Parallel Processing, 1988, pp. 1–6.

[11] J. Brandenburg and D. Scott, *Embedding of communication trees and grids into hypercubes*, Tech. Rep. 280182-001, Intel Scientific Computers, 1985.

[12] M. Chan, *Dilation-2 embeddings of grids into hypercubes*, in Proceedings International Conference on Parallel Processing, 1988, pp. 295 – 298.

[13] W.-K. Chen, *A graph-oriented mapping strategy for a hypercube*, Master's thesis, North Carolina State University, 1988.

[14] W.-K. Chen and E. Gehringer, *A graph-oriented mapping strategy for a hypercube*, in Proceedings Third Conference on Hypercube Concurrent Computers and Applications, 1988,

pp. 200 – 209.

[15] W.-K. Chen and M. Stallmann, *Local search variants for hypercube embedding*, in Proceedings Fifth Distributed Memory Computing Conference, 1990. To appear.

[16] G. Cybenko, D. Krumme, and K. Venkataraman, *Fixed hypercube embedding*, Information Processing Letters, 25 (1987), pp. 35 – 39.

[17] F. Ercal, J. Ramanujam, and P. Sadayappan, *Task allocation onto a hypercube by recursive mincut bipartitioning*, Journal of Parallel and Distributed Computing, (1990). To appear.

[18] F. Ercal and P. Sadayappan, *One-to-one mapping process graphs onto a hypercube*, in Proceedings Supercomputing '89, ACM, 1989, pp. 91 – 98.

[19] C. Fiduccia and R. Mattheyses, *A linear-time heuristic for improving network partitions*, in Proceedings 19th Design Automation Conference, 1982, pp. 175 – 181.

[20] K. Fukunaga, S. Yamada, and T. Kasai, *Assignment of job modules onto array processors*, IEEE Transactions on Computers, C-36 (1987), pp. 888 – 891.

[21] A. Gabrielian and D. Tyler, *Optimal object allocation in distributed computer systems*, in Proceedings International Conference on Distributed Computer Systems, 1984, pp. 88 – 95.

[22] M. Garey and R. Graham, *On cubical graphs*, Journal of Combinatorial Theory, 18 (1975).

[23] F. Heath, *Origins of the binary code*, Scientific American, 227 (1972), pp. 76 – 83.

[24] W. Hillis, *The Connection Machine*, MIT Press, 1985.

[25] J. Hong, K. Mehlhorn, and A. Rosenberg, *Cost trade-offs in graph embeddings with applications*, J. Assoc. Comput. Mach., 30 (1983), pp. 709 – 728.

[26] Intel Scientific Computers, *Direct-Connect^{tm} routing solves node communications challenge*, iSC Currents, (1987), pp. 5–6.

[27] D. S. Johnson, C. R. Aragon, L. A. McGeogh, and C. Schevon, *Optimization by simulated annealing: An experimental evaluation (part I)*. Typescript.

[28] B. Kernighan and S. Lin, *An efficient heuristic procedure for partitioning graphs*, Bell System technical Journal, (1970), pp. 291 – 307.

[29] S. Kirkpatrick, C. Gelatt, Jr., and M. Vecchi, *Optimization by simulated annealing*, Science, (1983), pp. 671 – 680.

[30] O. Krämer and H. Mühlenbein, *Mapping strategies in message-based multiprocessor systems*, Parallel Computing, 9 (1989), pp. 213 – 225.

[31] S.-Y. Lee and J. Aggarwal, *A mapping strategy for parallel processing*, IEEE Transactions on Computers, C-36 (1987), pp. 433 – 442.

[32] D. Lenoski, J. Laudon, A. G. Kourosh Gharachorloo, and J. Hennessy, *The directory-based cache coherence protocol for the dash multiprocessor*, in Proceedings of 17th International Symposium on Computer Architecture, May 1990. To appear.

[33] B. Monien and I. Sudborough, *Simulating binary trees on hypercubes*, in VLSI Algorithms and Architectures: 3rd Aegean Workshop on Computing, Lecture Notes in Computer Science 319, Springer Verlag, 1988, pp. 170 – 180.

[34] H. Mühlenbein, M. Gorges-Schleuter, and O. Krämer, *New solutions to the mapping problem of parallel systems: The evolution approach*, Parallel Computing, 4 (1987), pp. 269 – 279.

[35] J.-L. Pazat, *Outils pour la Programmation d'un Multiprocesseur à Mémoires Distribuées*, PhD thesis, Université de Bordeaux I, February 1989.

[36] J. Ramanujam, F. Ercal, and P. Sadayappan, *Task allocation by simulated annealing*, in Proceedings International Conference on Supercomputing, 1988.

[37] Y. Saad and M. H. Schultz, *Topological properties of hypercubes*, IEEE Transactions on Computers, C-37 (1988).

[38] P. Sadayappan and F. Ercal, *Nearest-neighbor mapping of finite element graphs onto processor meshes*, IEEE Transactions on Computers, C-36 (1987), pp. 1408 – 1424.

[39] K. Schwan and C. Gaimon, *Automating resource allocation for the Cm* multiprocessor*, in Proceedings International Conference on Distributed Computer Systems, 1984, pp. 310 – 320.

[40] L. W. Tucker and G. G. Robertson, *Architecture and applications of the Connection Machine*, IEEE Computer, 21 (1988), pp. 26–38.

[41] A. WAGNER, *Embedding arbitrary binary trees in a hypercube*, Journal of Parallel and Distributed Computing, 7 (1989), pp. 503–520.

[42] A. WAGNER AND D. CORNEIL, *Embedding trees in the hypercube is NP-complete*, SIAM Journal on Computing, 19 (1990), pp. 570 – 590.

[43] A. WU, *Embedding of tree networks into hypercubes*, Journal of Parallel and Distributed Computing, 2 (1985), pp. 238 – 249.