Computer Science Department Technical Report
University of California
Los Angeles, CA 90024-1596

# MULTI-WAY NETLIST PARTITIONING USING SPACEFILLING CURVES

C. Alpert
A. Kahng

June 1993
CSD-930016

# Multi-Way Netlist Partitioning Using Spacefilling Curves

Charles J. Alpert and Andrew B. Kahng

UCLA Computer Science Dept. Los Angeles, CA 90024-1596

## Abstract

Spectral geometric embeddings of a circuit netlist can lead to fast, high quality multi-way partitionings [1] [22]. In this work, we introduce a new and effective class of geometric partitioning algorithms. Our approach first orders the (spectrally) embedded netlist modules using a d-dimensional *spacefilling curve*. Then, subject to the constraint that clusters must lie contiguously along the spacefilling curve, dynamic programming techniques can obtain *optimal* multi-way partitioning solutions in low-order polynomial time for a range of objective functions including Scaled Cost [22]. The technique easily accomodates both upper and lower bounds on cluster size. Experimental results show excellent results, especially for $k \leq 4$.

## 1 Preliminaries

In top-down layout synthesis of large VLSI systems, the goal of partitioning is to reveal the *natural* circuit structure, via a decomposition into $k$ subcircuits with minimum connectivity between the subcircuits. A generic problem statement is as follows:

**General $k$-Way Partitioning:** Given a circuit netlist $G = (V, E)$ with $n$ modules in $V = \{v_1, v_2, \ldots, v_n\}$, construct a *$k$-way partitioning*, denoted as $P^k$, which divides $V$ into $k$ disjoint *clusters* $C_1, C_2, \ldots, C_k$ to minimize a given objective function $f(P^k)$.

In this work, we consider the *small-k partitioning* (SKP) regime where $k \ll n$, e.g., $k \leq 10$ for $n \geq 1000$.

The SKP problem arises in high-level VLSI system partitioning and floorplanning, as well as in classification analysis. Early approaches in the VLSI realm have involved seeded epitaxial growth, extensions of the Fiduccia-Mattheyses iterative interchange bipartitioning algorithm [19], and a primal-dual iteration motivated by a generalization of the minimum ratio cut metric [20]. Recently, Yeh et al. [21] proposed a "shortest-path clustering" (SPC) method, where "shortest paths" between random pairs of modules are iteratively deleted from the netlist graph until there are $k$ connected components (i.e., the clusters). The algorithm probabilistically captures the relationship between multicommodity flow and minimum ratio cut, and yields high-quality solutions when measured by *cluster ratio*, which Yeh et al. characterize as the "proper" $k$-way generalization of the ratio cut objective.

**Minimum Cluster Ratio SKP:** Find a partitioning $P^k = \{C_1, C_2, \ldots, C_k\}$, $2 \le k \le |V|$, that minimizes

$$f(P^k) = \frac{c(P^k)}{\sum_{i=1}^{k-1} \sum_{j=i+1}^{k} |C_i| \times |C_j|}$$

where $c(P^k)$ is the number of nets which cross between two or more clusters in $P^k$.

Other SKP approaches extend well-established spectral methods. In 1970, Hall [12] generated one-dimensional netlist embeddings from the eigenvector corresponding to the second smallest eigenvalue $\lambda_2$ of the netlist Laplacian $Q = D - A$ (where $D$ is the diagonal degree matrix and $A$ is the adjacency matrix). This embedding corresponds to a module placement $\vec{x}$ along the real line which minimizes total squared wirelength subject to the condition $\vec{x} \cdot \vec{x}^T = 1$.

Hagen and Kahng [11] established a connection between $\lambda_2$ and optimal ratio cuts by showing that $\frac{\lambda_2}{n}$ is a tight lower bound for ratio cut cost; this motivated their method of generating high-quality ratio cut bipartitionings by "splitting" the eigenvector. In [22], Zien generalizes the result of [11] to $k$-way ratio cut partitioning, using the first $k$ eigenvectors of the netlist Laplacian to construct an orthogonal *projector* which maps an $n$-dimensional space into a $k$-dimensional space (the paper of Chan et al. [6] is an abridged version of [22]). Ideally, the $n$ elementary unit vectors in the $n$-space (the modules) will be mapped to exactly $k$ distinct points in the $k$-space (the clusters) by this projector. Since this does not occur in practice, the authors of [22] use a heuristic based on directional cosines to obtain a $k$-way partitioning of the modules embedded in $k$-space. Zien's approach requires additional matrix manipulations and a more complicated netlist-based partitioning methodology in comparison with our methods below.

The authors of [22] propose the dimensionless *Scaled Cost* metric as a multi-way generalization of the ratio cut objective:

**Minimum Scaled Cost SKP:** Find a partitioning $P^k = \{C_1, C_2, \ldots, C_k\}$, $2 \le k \le |V|$, that minimizes

$$f(P^k) = \frac{1}{n(k-1)} \sum_{i=1}^{k} \frac{E_i}{|C_i|}$$

where $E_i$ is the number of signal nets crossing the boundary of the cluster $C_i$. Thus, both [21] and [22] propose generalizations of the ratio cut concept. Each metric is robust, and automatically accounts for both cut nets and size balance among the clusters. However, both objectives are easily shown NP-Complete by restriction to minimum ratio cut.

Recently, Alpert and Kahng [1] proposed the KCenter heuristic, which computes spectral geometric embeddings of the netlist and then applies simple, *geometric* partitioning algorithms to obtain fast partitioning solutions. While this idea goes at least as far back as the 1970 work of Hall, [1] demonstrated that proper choices for the *net model*, for the partitioning *objective*, and for the partitioning *algorithm* are all critical to success. In [1], geometric embeddings were generated using a "partition-specific" clique net model which assigns cost $\frac{4}{p \cdot (p-1)}$ to each edge in the clique that represents a $p$-pin net. This weighting scheme ensures that a cut of a large net will have the same expected cost as a cut of a small net. As in [22], the $d$-dimensional geometric netlist embedding is generated directly from the eigenvectors corresponding to the $d$ smallest nonzero eigenvalues of the Laplacian of the resulting graph representation.

Let $d(v_i, v_j)$ denote the Euclidean distance between (the geometric embeddings of) $v_i, v_j \in V$. For clusters $C_1$ and $C_2$, two cluster measures are the cluster diameter, $diam(C_1) = \max_{v_i, v_j \in C_1} d(v_i, v_j)$, and the split between two clusters, $split(C_1, C_2) = \min_{v_i \in C_1, v_j \in C_2} d(v_i, v_j)$. Alpert and Kahng [1] surveyed the following standard objectives in partitioning points of the geometric embedding,

**Formulation 1:** *Max-Split Partitioning.* Maximize

$$f(P^k) = \min_{1 \leq i < j \leq k} \{split(C_i, C_j)\}$$

**Formulation 2:** *Min-Diameter Partitioning.* Minimize

$$f(P^k) = \max_{1 \leq i \leq k} \{diam(C_i)\}$$

**Formulation 3:** *Min-Sum-Diameters Partitioning.* Minimize

$$f(P^k) = \sum_{1=i}^{k} diam(C_i)$$

**Fact 1:** Formulation 1 can be solved optimally by the Single-Linkage Algorithm [13].

However, Single-Linkage yields poor results in practice [1], implying that Formulation 1 does not correspond well to netlist partitioning. Three other results show that finding the best partitioning with respect to any diameter-related criterion is as intractable as minimizing Cluster Ratio or Scaled Cost.

**Fact 2:** Formulations 2 and 3 are NP-Complete for $k \geq 3$ and $d \geq 2$ [16].

**Fact 3:** Solving Formulation 2 within a factor $< 2$ of optimal is NP-complete for $d \geq 3$ [7].

**Fact 4:** In general graphs whose edge weights do not satisfy the triangle inequality, neither Formulation 2 nor Formulation 3 may be approximated to within any fixed constant factor of optimal for $k \geq 3$ [8].

After extensive experiments, [1] showed that Formulation 2 tends to yield the best netlist partitioning results; that work also proposed use of the KCenter algorithm [8], which in $O(n \log k)$ time guarantees a solution to Formulation 2 within a factor of two of optimal. The moderate success of KCenter for VLSI partitioning suggests that the spectral geometric embedding indeed preserves fundamental properties of the netlist, i.e., two modules that are strongly connected in the netlist will be close to each other in the geometric embedding. However, the approach of [1] has obvious shortcomings. Minimizing the maximum cluster diameter has only a loose correlation to either Scaled Cost or Cluster Ratio. Furthermore, KCenter solutions for the min-diameter objective are not as good as those obtained by higher-complexity diameter partitioning algorithms. In general, the most important observation is that a strictly geometric formulation will always be handicapped since it ignores valuable information in the netlist. The geometric embedding should serve as a *guide*, rather than an absolute; this motivates our new approach which utilizes geometric and netlist information simultaneously when making partitioning choices.

The remainder of this paper is organized as follows. In Section 2, we present a new *Restricted Partitioning* (RP) formulation and outline the approach which yields efficient, optimal solutions. In Section 3, we explore spacefilling curves as a method of generating instances of Restricted Partitioning from a given $d$-dimensional netlist embedding. In Section 4, we present dynamic programming algorithms, along with natural extensions, which optimally solve a number of RP variants. Section 5 presents experimental results showing that our new approach substantially improves over previous methods and Section 6 concludes with directions for future research.

## 2 A New Approach to Partitioning

The genesis of our approach lies in the Traveling Salesman Problem (TSP) heuristic of Karp [14], which uses a good partitioning of a planar pointset to construct a good TSP tour. The heuristic tour visits the clusters of the partitioning one at a time, visiting every point in each cluster before moving to the next cluster. We ask whether an "inverse" methodology can succeed, i.e., whether we can use a "good tour" of the geometric pointset to generate a good partitioning (ineed, if each cluster of a partitioning is a contiguous "slice" of some tour, then Karp's algorithm when applied to this partitioning could recover the tour). Our new partitioning heuristic constructs a "natural" circular *permutation* or "tour" of the modules, i.e., a bijection $\Pi : V \rightarrow V$. If we write $\Pi(v_i) = v_{\pi_j}$; then $\Pi$ can be written using the ordered-set notation $\{v_{\pi_1}, v_{\pi_2}, \ldots, v_{\pi_n}\}$. A *slice* $[i,j]$ of $\Pi$ is a contiguous subset of modules $\{v_{\pi_1}, v_{\pi_2}, \ldots, v_{\pi_n}\}$; we treat indices modulo $n$ so that $[i,j] = \{v_{\pi_i}, v_{\pi_{i+1}}, \ldots, v_{\pi_j}\}$ if $i \leq j$ and $[i,j] = \{v_{\pi_j}, v_{\pi_{j+1}}, \ldots, v_{\pi_n}\} \cup \{v_{\pi_1}, v_{\pi_2}, \ldots, v_{\pi_i}\}$ if $i > j$. Instead of partitioning the pointset corresponding to a $d$-dimensional netlist embedding, we partition $\Pi$ (a 1-dimensional permutation of the pointset) such that each cluster is a slice. We therefore obtain the following "restricted" $k$-way partitioning problem:

**Restricted $k$-Way Partitioning (RP):** Given a permutation $\Pi : V \rightarrow V$, cluster size bounds $L$ and $U$, and an objective $f$, construct $P^k = \{C_1, C_2, \ldots, C_k\}$ which optimizes $f(P^k)$ such that the following conditions hold:

**Condition 1:** $\forall v_{\pi_i} \in V$, $v_{\pi_i} \in C_j$ for exactly one $j, 1 \leq j \leq k$

**Condition 2:** if $v_{\pi_i}, v_{\pi_j} \in C$ for some cluster $C$ and $i \leq j$, then either

**(a)** $[i,j] \subseteq C$ or

**(b)** $[j,i] \subseteq C$.

**Condition 3:** $L \leq |C_j| \leq U \quad \forall C_j \in P^k, 1 \leq j \leq k$.

If we set $L = 1$ and $U = n$, Condition 2 is the difference between RP and the general SKP formulation. Again, one can think of the permutation as a *tour* of the modules, with the $k$-way partitioning constructed by removing $k$ edges from the tour. The main advantage of RP is that dynamic programming yields *optimal* $k$-way solutions for a large class of objectives (including Scaled Cost and diameter-related objectives) in low-order polynomial time. This is a big win since the general Scaled Cost and diameter formulations are NP-Complete for $k > 2$. User-imposed bounds on cluster size are also handled transparently, with optimality being retained without any increase in time complexity. If we further restrict the RP formulation by removing Condition 2(b), the permutation is a *linear ordering* which narrows the solution space but allows an $O(n)$ factor speedup. As shown in Section 4, the best RP solutions yield excellent $k$-way netlist partionings despite the ordering constraint: results for 2-way partitioning of the SIGDA Layout Synthesis benchmarks result in an *average* of 45% improvement over the best previous spectral approaches.

## 3 Spacefilling Curves in $d$ Dimensions

Given a geometric pointset (the $d$-dimensional module embedding), we seek to construct a permutation $\Pi$ which adequately preserves proximity in the netlist. Recall that two modules which are strongly connected in the netlist will tend to be near each other in the spectral geometric embedding. For the RP approach to be successful, strongly connected modules must remain near each other inco $\Pi$. Furthermore, if two strongly connected modules are *not* adjacent in the permutation, they should be separated by modules with which they may profitably share a cluster.

From the intuition above, a good TSP solution over the embedded pointset should suffice since it is unlikely to wander out of, and then back into, a natural cluster. However, it is not obvious what TSP heuristic to use. For example, even relatively good solutions such as the greedy nearest-neighbor approach [15] can yield long edges which would force unrelated modules to be adjacent to each other in the permutation.

Bartholdi and Platzman have used spacefilling curves to provide a provably good TSP heuristic [3]. They use the construction due to Sierpinski (1912), the 2-dimensional case of which is illustrated in Figure 1. In the Figure, the successive approximations become more and more refined until the curve "fills" up the unit square (assuming discrete intervals), i.e., it passes over every point. The order in which points are visited by the curve yields the desired permutation. Figure 2 shows the spacefilling curve ordering for (a) a uniformly random set of 150 points in the plane, and (b) a 2-dimensional embedding of the Primary1 layout synthesis benchmark. Notice that any $d$-dimensional spacefilling curve can be represented as a mapping from 1-dimensional space (the curve) into $d$-dimensional space (the unit $d$-cube). Such curves as the 2-dimensional Sierpinski construction can be easily extended into higher dimensions by recursively applying the 2-dimensional mapping function.

[3] observed that this heuristic yields planar TSP tours within 25% of optimal in practice and showed a constant-factor expected error for uniformly random instances. The heuristic also has an $\Theta(\log n)$ worst-

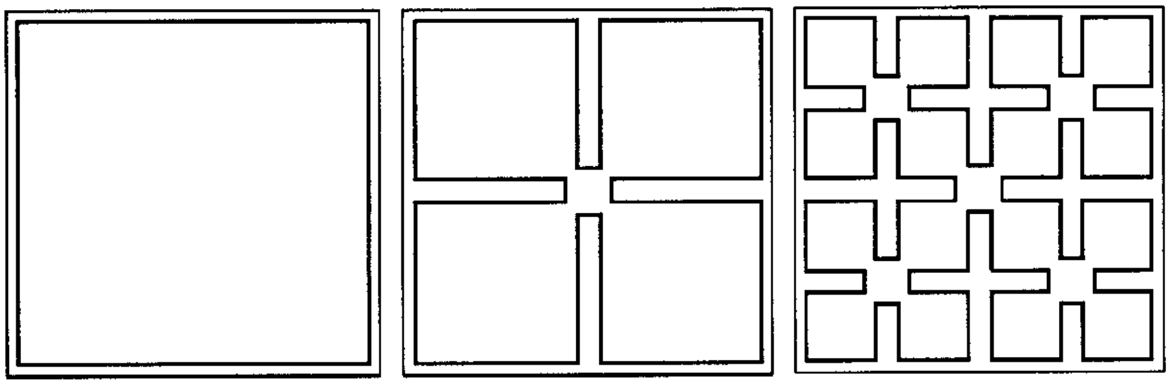

Figure 1: The Sierpinski spacefilling curve in the plane is the limit of a sequence of resursive constructions.

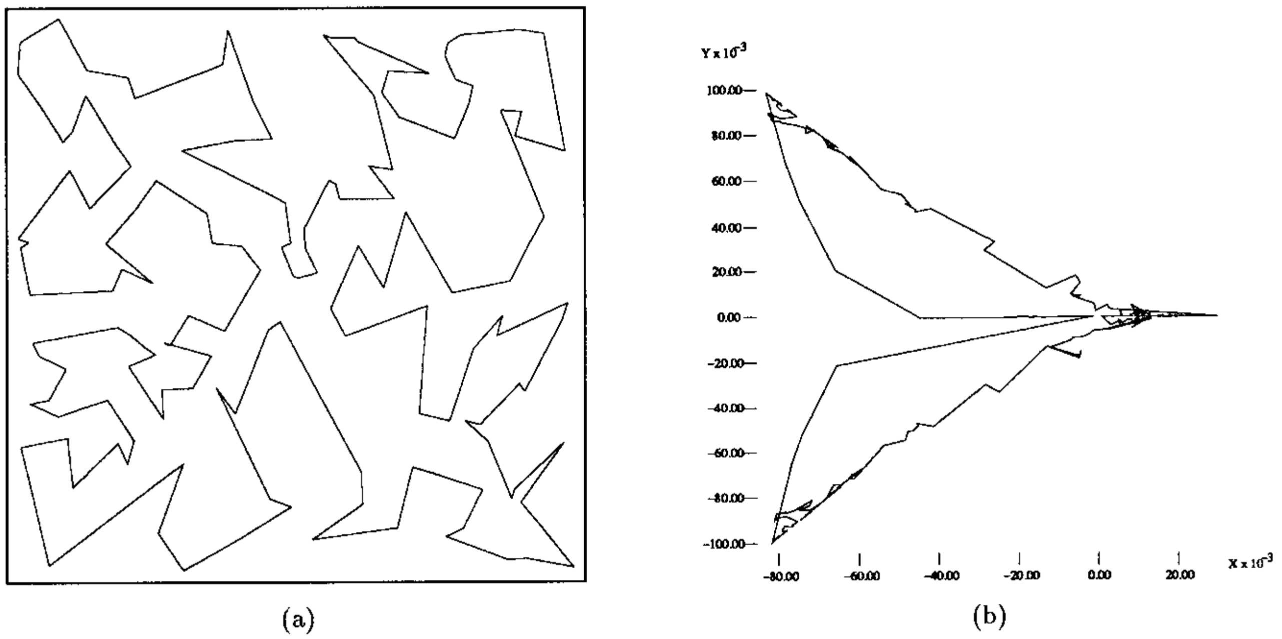

Figure 2: The ordering generated by a spacefilling curve for (a) 150 random points in the plane, (b) a 2-dimensional embedding of Primary 1.

case error bound [3] [5]. The Sierpinski curve is certainly not the only spacefilling curve which can be used. Figure 3 shows other recursive constructions, including two that yield linear orderings as opposed to circular orderings. The authors of [4] empirically found that the Sierpinski curve outperforms these and other spacefilling curves, especially for uniformly random point locations. They also note that other spacefilling curves might be considered for nonuniform distributions and even outline a method for creating application-specific spacefilling curves.

The Sierpinski curve seems to suit our purposes since points which are close to each other in the geometry will generally also be close along the curve. More critically, the Sierpinski curve avoids inducing long edges;

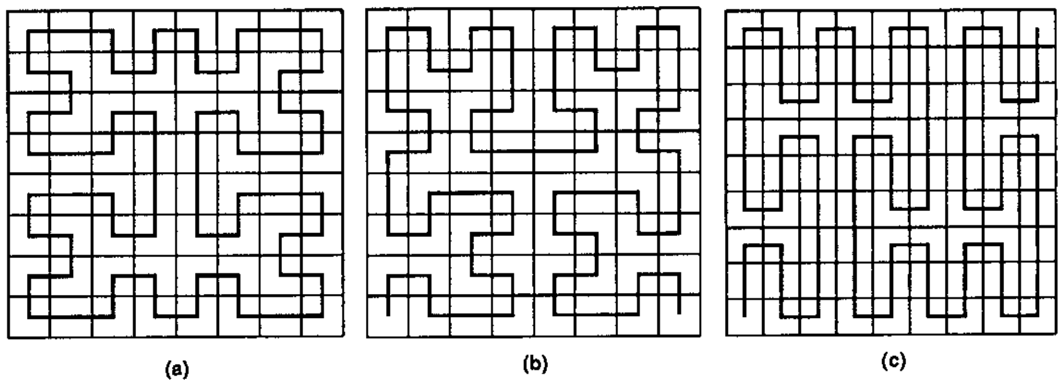

Figure 3: Alternative constructions: (a) illustrates a closed spacefilling curve, while (b) and (c) are paths.

thus, if two points are close on the curve, they *must* be close to each other in geometry. However, the curve is also imperfect in that it explores one orthant entirely before moving on to the next. Thus, two points that are close together, but in different orthants, may be widely separated in the ordering.

A tremendous advantage of the Sierpinski curve is that its induced ordering can be calculated in just $O(n \log n)$ time. A spacefilling curve can be viewed as mapping the unit interval into $d$-dimensional space, hence computing $\Pi$ requires us to compute the *inverse* of this mapping function. In other words, we seek a function $\Theta : [0,1)^d \rightarrow [0,1)$ which assigns each point in the unit $d$-cube to a value in $[0,1)$. $\Theta$ must preserve the order in which points are visited by the curve such that $\Theta(v_i) \leq \Theta(v_j)$ if and only if $v_i$ appears before $v_j$ when traversing the curve (the "beginning" of the curve is the point $u$ which minimizes $\Theta(u)$ and the "end" of the curve is the point $v$ which maximizes $\Theta(v)$). For the Sierpinski curve, the function $\Theta$ can be evaluated at any point in constant time, where the constant depends on the depth of the recursion needed for the curve to "resolve" the entire space (such that the positions of all points on the curve are distinguishable). After $\Theta(v_i)$ has been computed for each $v_i \in P$, the $\Theta$ values are sorted to yield the permutation $\Pi$. Thus, $\Pi$ can be computed in $O(n \log n)$ time overall.

In Figure 4, we reproduce from [4] the calculation of $\Theta$ for completeness and also to show its simplicity. Step 7 is not included in [4] and we believe it was omitted as an oversight. We number the $2^d$ orthants from 0 to $2^d - 1$, corresponding to the order in which they are visited by the Sierpinski curve. This numbering can be represented by a $d$-bit Grey code since each orthant will be adjacent to the previous one; specifically, we use the Grey code which flips the rightmost bit possible without repeating an earlier sequence. In Figure 4, the function *Grey* accepts an orthant and returns a value $Q$ between 0 and $2^d - 1$, i.e., *Grey* $(0\ldots0) = 0$, *Grey* $(0\ldots01) = 1$, *Grey* $(0\ldots011) = 2$, *Grey* $(0\ldots010) = 3$, *Grey* $(0\ldots0110) = 4, \ldots,$ *Grey* $(10\ldots0) = 2^d - 1$. The function **Theta** itself has arguments $X$ (a $d$-dimensional point) and *depth* (a variable which specifies the minimum number of bits needed to distinguish any two points, i.e., *depth* indicates the granularity of the pointset locations in the $d$-dimensional unit cube).

```
Function Theta ($X, depth$)
Input:   point in $\Re^d = X$ :array $[1 \ldots d]$ of real
         $depth$ - granularity measure
Output:  $\Theta \equiv$ real number in $[0,1)$ which indicates place on curve
Vars:    Temporary $d$-dimensional point $Y$
         Orthant number $Q$
         Recursive subsolution - $SubTheta$
1. $C = Grey(1 \ldots 1)/2^d$
2. if ($depth = 0$ or $X = (1 \ldots 1)$) then return $C$
3. for $i = 1$ to $d$ do
       $Y[i] = \min\{\lfloor 2 \cdot X[i] \rfloor, 1\}$
4. $Q = Grey(Y[1] \ldots Y[d])$
5. for $i = 1$ to $d$ do
       $Y[i] = 2 \cdot |X[i] - 0.5|$
6. $SubTheta$ = Theta($Y, depth - 1$)
7. if ($Q$ mod $2 = 1$) then $SubTheta = 1 - SubTheta$
8. $Num = (Q + SubTheta - C)/2^d$
9. return $Num - \lfloor Num \rfloor$
```

Figure 4: Computation of $\Theta$

## 4 A Dynamic Programming RP Solution

Given an ordering $\Pi$ over the pointset, dynamic programming efficiently finds optimal RP solutions for a variety of objective functions $f$. Let $w(C_i)$ be the cost of having cluster $C_i$ in our partitioning $P^k$. For dynamic programming to yield optimal solutions, it is necessary that all partial subsolutions also be optimal, e.g., if $P^3 = \{C_1, C_2, C_3\}$ is optimal then $\{C_1, C_2\}$ must be the optimal partitioning of the modules in $\{C_1 \cup C_2\}$. This *principle of optimality* requires $f$ to be *monotone nondecreasing* over $w$, i.e., for any $P^k = \{C_1, C_2, \ldots, C_k\}$ and $Q^k = \{C'_1, C'_2, \ldots, C'_k\}$ with $w(C_i) \leq w(C'_i)$ for $1 \leq i \leq k$, $f$ is monotone nondecreasing if and only if $f(P^k) \leq f(Q^k)$. Common choices for $f$ amenable to dynamic programming involve maximum and summation. For example:

- Maximum cluster diameter objective:

$$f(P^k) = \max_{1 \leq i \leq k} w(C_i) \text{ with } w(C_i) = diam(C_i)$$

- Sum of Diameters objective:

$$f(P^k) = \sum_{1 \leq i \leq k} w(C_i) \text{ with } w(C_i) = diam(C_i)$$

- Scaled Cost objective:

$$f(P^k) = \frac{1}{n(k-1)} \sum_{C_i \in P^k} w(C_i) \text{ with } w(C_i) = \frac{E_i}{|C_i|}$$

Three key observations allow dynamic programming to obtain efficient, optimal RP solutions for any $k$.

Any cluster is a slice of the permutation and is uniquely determined by the first and last elements in the slice. Thus:

**Observation 1:** *There are only $(U + L - 1)n$ clusters which can possibly be part of an RP solution.*

The cluster corresponding to slice $[i,j]$ is denoted by $C_{[i,j]}$ and we let $P^k_{[i,j]}$ denote a $k$-way RP solution over the slice $[i,j]$. The optimal $k$-way RP solution over $[i,j]$ is $\hat{P}^k_{[i,j]}$. Notice that we always have $P^1_{[i,j]} = \hat{P}^1_{[i,j]} = \{C_{[i,j]}\}$. In general we will use the set of $\hat{P}^k_{[i,j]}$ partitionings as "building blocks" for solutions of the form $\hat{P}^{k'}_{[i',j']}$ where $[i,j] \subset [i',j']$ and $k < k'$.

**Observation 2:** *$\hat{P}^k_{[i,j]}$ can be expressed as $\hat{P}^{k-1}_{[i,m]} \cup \{C_{[m+1,j]}\}$ for some $m$ such that $L \leq |C_{[m+1,j]}| \leq U$.* Observation 2 follows from the principle of optimality above, whereby each subset of contiguous clusters in $\hat{P}^k$ is an optimal solution over the slice covered by the subset, e.g., if $P^k_{[i,m]} \subset \hat{P}^{k'}_{[i,j]}$ then $P^k_{[i,m]}$ must be optimal over $[i,m]$. To solve RP, we precompute the cost of every cluster (recall there are at most $(U+L-1)n$ clusters); these give all optimal 1-way partitioning subsolutions. We then build 2-way partitioning solutions $\hat{P}^2_{[i,j]}$ from the $\hat{P}^1_{[i,j]}$, etc. until an optimal $k$-way solution is derived. The template of Figure 5 outlines this generic dynamic programming solution to RP, which we call DP-RP. Note that the template assumes the existence of a procedure Calculate_Cluster_Costs, which computes $w(C)$ for all possible clusters. The next subsection gives versions of this procedure for specific examples of $w$.

```
DP-RP Generic RP Algorithm  $(\Pi, L, U)$
Input:   Permutaion $\Pi = \{v_{\pi_1}, v_{\pi_2}, \ldots, v_{\pi_n}\}$
         Lower and upper cluster size bounds $L, U$
Output:  Optimal RP solution $\hat{P}^k$
Vars:    Subsolutions $\hat{P}^{k'}_{[i,j]}$
         Index $k'$ denoting current partitioning size
         Index $m$ marking possible cluster to merge into partitioning
1. Generate $f(\hat{P}^1_{[i,j]}) = w(C_{[i,j]})$ from Calculate_Cluster_Costs
2. for $k' = 2$ to $k$ do
3.   for each $i, j$ do
4.     $f_{best} = \infty$
5.     for $m = j - U$ to $j - L$ do
6.       if $f_{best} < f(\hat{P}^{k'-1}_{[i,m]} \cup \{C_{[m+1,j]}\})$ then
7.         $f_{best} = f(\hat{P}^{k'-1}_{[i,m]} \cup \{C_{[m+1,j]}\})$, $\hat{P}^{k'}_{i,j} = \hat{P}^{k'-1}_{[i,m]} \cup C_{[m+1,j]}$
8. return $f(\hat{P}^k) = \min_{1 \leq i \leq n} f(\hat{P}^k_{[i,i-1]})$
```

Figure 5: DP-RP algorithm

**Observation 3:** *DP-RP has complexity $O(k(U - L)n^2)$ assuming that Calculate_Cluster_Costs has $O(n^2)$ complexity.* When there are no cluster size constraints, DP-RP has $O(kn^3)$ complexity.

We will now present $O(n^2)$ algorithms for the Calculate_Cluster_Costs procedure for two different cluster cost measures.

## 4.1 Calculating Cluster Costs

When $w(C_i) = diam(C_i)$, Figure 6 gives a simple $O(nU)$ implementation of Calculate_Cluster_Costs (Diameter) [1].

```
Calculate_Cluster_Costs (Diameter)  $(\Pi, L, U)$
Input:   Permutation $\Pi = \{v_{\pi_1}, v_{\pi_2}, \ldots, v_{\pi_n}\}$
         Lower and upper cluster size bounds $L, U$
Output:  $w(C_{[i,j]}) = diam(C_{[i,j]})$ for every possible cluster $C_{[i,j]}$
Vars:    $\Delta$ - one less than size of current clusters
1. for $1 \le i \le n$ do $w(C_{[i,i]}) = 0$
2. for $\Delta = 1$ to $U - 1$ do
3.    for $i = 1$ to $n$ do
4.       $j = ((i + \Delta - 1) \bmod n) + 1$
5.       $w(C_{[i,j]}) = \max\{w(C_{[i,j-1]}), w(C_{[i+1,j]}), d(v_{\pi_i}, v_{\pi_j})\}$
```

Figure 6: Calculate_Cluster_Costs (Diameter)

The algorithm starts with clusters $C_{[i,i]}$ of diameter zero. The key observation is that any edge within cluster $C_{[i,j]}$ is contained in $C_{[i+1,j]}$, or is contained in $C_{[i,j-1]}$, or is edge $e_{ij}$ itself. Thus, Step 5 obtains the diameter of each new cluster in constant time, yielding the $O(nU)$ complexity bound. While our usage assumes Euclidean distances, the procedure may be applied to any dissimilarity measure $d$.

Our next example is $w(C_i) = \frac{E_i}{|C_i|}$, which pertains to the Scaled Cost metric (discussed in Section 1). This is a more complicated computation since it involves topological information from the netlist. Figure 7 illustrates the calculation of all cluster costs for this $w(C_i) = \frac{E_i}{|C_i|}$ in $O(nU)$ time, if we assume that the number of signal nets is $O(n)$. (This is valid since both fanout and cell I/O are bounded for any given technology; in practice, the number of nets is actually very close to $n$.)

Steps 1-9 calculate the outdegree for each cluster, and $w(C_{[i,j]})$ is computed in Step 10. Given the value of $w(C_{[i,j-1]})$, one can compute $w(C_{[i,j]})$ by adding $v_{\pi_j}$ to cluster $C_{[i,j-1]}$ and checking whether any cut nets become completely contained in the cluster (Step 8), or whether any previously uncut nets become cut (Step 8). Of course, the desired $O(nU)$ time complexity hinges on Steps 8 and 9 being executed in constant time. We accomplish this by maintaining an array *Cut_Sigs* over the signal nets, where $Cut\_Sigs[s] = 1$ if $s$ is cut by our current cluster $Cut\_Sigs[s] = 0$ otherwise. In Step 2, we start with the initial cluster $C_{[i,i]}$ and set $Cut\_Sigs[s] = 1$ if $s$ contains module $v_{\pi_i}$, and $Cut\_Sigs[s] = 0$ otherwise. We also keep a counter $Count[s]$, which records the number of modules connected by net $s$ in the current cluster, and an array $Net\_Sizes[s]$, which holds the total number of modules in $s$ (NetSizes can be computed in the initialization). To execute Step 8, we verify the **if** condition using the *Count* array. If $Count[s] = Net\_Sizes[s]$, then $s$ is completely contained in the current cluster, and we set $Cut\_Sigs[s] = 0$ since this net is no longer cut. To execute Step 9, we check that $Cut\_Sigs[s] = 0$ in which case adding $v_{\pi_j}$ to the current cluster will cause $s$ to be cut. For these newly cut nets we set $Cut\_Sigs[s] = 1$ and $Count[s] = 1$.

[1]The general modulo manipulation of Step 4 appears throughout this report. Addition of indices are done in this way in order to keep indices within the range $1 \ldots n$.

```
Calculate_Cluster_Costs (Scaled Cost)   $(\Pi, L, U)$
Input:    Permutation $\Pi = \{v_{\pi_1}, v_{\pi_2}, \ldots, v_{\pi_n}\}$
          Lower and upper cluster size bounds $L, U$
Output:   $w(C_{[i,j]} = \frac{E_{[i,j]}}{|C_{[i,j]}|}$ for every possible cluster $C_{[i,j]}$
Vars:     $\Delta$ - one less than size of current clusters
          $S_i$ - set of signal nets which contain module $v_{\pi_i}$
1. for $i = 1$ to $n$ do
2.   $w(C_{[i,i]}) = |S_i|$ where $S_i = \{s|\text{signal net } s \text{ contains module } v_{\pi_i}\}$
3.   for $\delta = 1$ to $U$ do
4.     $j = ((i + \delta - 1) \bmod n) + 1$
5.     $S_j = \{s|\text{signal net } s \text{ contains module } v_{\pi_j}\}$
6.     $w(C_{[i,j]}) = w(C_{[i,j-1]})$
7.     for every $s \in S_j$ do
8.       if ($s$ is completely contained in $C_{[i,j]}$) then decrement $w(C_{[i,j]})$
9.       if ($s$ contains no modules of $C_{[i,j-1]}$) then increment $w(C_{[i,j]})$
10. for every $w(C_{[i,j]})$ calculated do $w(C_{[i,j]}) = \frac{w(C_{[i,j]})}{j-i+1}$
```

Figure 7: Calculate_Cluster_Costs (Scaled Cost)

Since each $S_j$ is calculated at most $U$ times, all the $S_j$ calculations are done in time $U$ times the size of the entire netlist (which is of the same order as the number of modules because of constant degrees). Since all operations are done in constant time, Calculate_Cluster_Costs for Scaled Cost runs in $O(nU)$ time.

## 4.2 Special Case: Optimizing Maximum Diameter

For certain choices of $f$ and $w$, invoking dynamic programming may not be necessary. We recognize a special case which enables a $\frac{n}{\log n}$ speedup, namely

$$f(P^k) = \max_{C_i \in P^k} w(C_i).$$

Here we will restrict ourselves to the case $L = 1$ (although this restriction is not required, extending this special case to encompass $L > 1$ is nontrivial). In this case, $f(P^k)$ can take on only a polynomial number of possible values: since there are at most $nU$ possible clusters, there can be at most $nU$ possible values for $f(P^k)$ despite an exponential number of possible partitionings $P^k$.

We ask the decision problem, "does there exist an RP solution $P^k$ with $f(P^k) \leq M$, for some value $M$". Given an oracle which solves this decision problem in time $O(T)$, we can solve the RP formulation by performing a binary search over the polynomial number of possible $f$ values, which requires a total of $O(T \log n + nU \log n)$ time.

In order to enable a significant speedup over our dynamic programming solution, it is also necessary that $w$ be monotone nondecreasing in the size of the cluster, i.e. if $[i,j] \subset [i',j']$, then $w(C_{[i,j]}) \leq w(C_{[i',j']})$. Thus $w = diam$ satisfies monotonicity while $w(C_i) = \frac{E_i}{|C_i|}$ does not. The monotonicity of $w$ allows us to greedily solve the decision question; we simply grow each cluster $C_i$ as large as possible as long as $w(C_i) \leq M$. The algorithm in Figure 8 solves the decision problem in $O(nU)$ time.

```
Decide_Cluster  $(\Pi, L, U, M)$
Input:    Permutation $\Pi = \{v_{\pi_1}, v_{\pi_2}, \ldots, v_{\pi_n}\}$
          Lower and upper cluster size bounds $L, U$
          Upper bound $M$
Output:   YES iff $\exists P^k$ such that $f(P^k) \leq M$
Vars:     Leftmost index $first$ of cluster $C_1$
          Current number of clusters in partitioning - $k'$
1. for $first = 1$ to $U$ do
2.   $i = j = first$
3.   repeat
4.     $k' = 1$
5.     while $((|C_{[i,j]}| \leq U)$ and $(w(C_{[i,j]}) \leq M)$ and $(j \neq ((first - 2) \bmod n) + 1))$ do
6.       increment $j$
7.     $C_{k'} = C_{[i,j-1]}$
8.     $i = j$
9.   until $(j = first$ or $k' = k)$
10.  if $(k' < k$ and $j = first)$ return YES
11.return NO
```

Figure 8: Decide_Cluster Algorithm

Decide_Cluster begins with $C_1 = \{v_{\pi_{first}}\}$ (initially $first = 1$) and traverses the ordering while constructing each cluster to be as large as possible. When a cluster violates either the size or cost constraint (failure of Step 5), Decide_Cluster stores the version of the cluster from just prior to the constraint violation (Step 7) and starts a new cluster. This continues until $k$ clusters are generated or every module has been placed into a cluster (Step 9); if the procedure has put every module into a cluster then a legal $P^k$ must exist (Step 10); otherwise, Decide_Cluster it tries again with the next starting point $v_{\pi_{first+1}}$. If all possible values for $first$ fail, then no partitioning exists with $f(P^k) \leq M$. We now prove the correctness of Decide_Cluster.

**Theorem:** Decide_Cluster returns YES if and only if there exists an RP solution such that $f(P^k) \leq M$.

**Proof:** If Decide_Cluster returns YES, $P^k$ is given by the $C_1, C_2, \ldots, C_k$ generated by the procedure (there may be fewer than $k$ clusters, but $k$ clusters can be generated without increasing $f$ by arbitrarily splitting large clusters). Assume that Decide_Cluster returns NO but there exists a partitioning $Q^k = \{C'_1, C'_2, \ldots C'_k\}$ with $f(Q^k) \leq M$. We assume that $C'_1$ has left endpoint $v_{\pi_{first}} \equiv v_{\pi_i}$ where $i$ is the smallest index of all left cluster endpoints in $Q^k$, and that the $C'_i$ are indexed in permutation order. Let $P^k = \{C_1, C_2, \ldots, C_k\}$ be the partitioning constructed by Decide_Cluster with this value for $first$.

**Inductive Hypothesis:** $|C'_1| + |C'_2| + \ldots |C'_{k'}| \leq |C_1| + |C_2| + \ldots |C_{k'}|$ for $1 \leq k' \leq k$.

**Basis ($k' = 1$):** Decide_Cluster constructs $C_1$ to be as large as possible while satisfying $|C_1| \leq U$ and $w(C_1) \leq M$. Since $C'_1$ also satisfies these constraints, $|C'_1| \leq |C_1|$.

**Induction:** Assume $|C'_1| + |C'_2| + \ldots |C'_{k'}| \leq |C_1| + |C_2| + \ldots |C_{k'}|$ holds for $k'$; we will show it holds for $k' + 1$. Assume otherwise, then we must have $C_{k'+1} \subset C'_{k'+1}$. But since $w(C'_{k'+1}) \leq M$, $C_{k'+1}$ can then be expanded while still satisfying $C_{k'+1} \subset C'_{k'+1}$ and $w(C'_{k'+1}) \leq M$ (since $w$ is monotone nondecreasing).

This is a contradiction since Decide_Cluster constructs all $C_i$ to be maximal.

Thus, if $Q^k$ is a legal RP solution, Decide_Cluster will return YES, proving the theorem. □

### 4.3 Improving Complexity: Linear Orders

So far, we have considered the RP formulation where both rules 2(a) and 2(b) apply. Disallowing 2(b) prevents wraparound, changing the tour into a linear ordering. While this restricts the solution space even further, we achieve a factor $n$ speedup from DP-RP. Figure 9 illustrates how DP-RP can be modified for linear orders with $O(kn(U-L))$ complexity, again assuming efficient implementation of the Calculate_Cluster_Costs procedure.

```
Linear Ordering Partitioning Algorithm
Input:   Linear ordering $\Pi = \{\pi_1, \pi_2, \ldots, \pi_n\}$
         Lower and upper cluster size bounds $L, U$
Output:  Optimal RP (without Condition 2(b)) solution $P^k$
Vars:    Subsolutions $\hat{P}^{k'}_{[i,j]}$
         Index $k'$ denoting current partitioning size
         Index $m$ marking possible cluster to merge into partitioning
1. Generate $f(\hat{P}^1_{[i,j]}) = w(C_{[i,j]})$ from Calculate_Cluster_Costs
2. for $k' = 2$ to $k$ do
3.   for every $j$ do
4.     $f_{best} = \infty$
5.     for $m = j - U$ to $j - L$ do
6.       if $f_{best} < f(\hat{P}^{k'-1}_{[1,m]} \cup \{C_{[m+1,j]}\})$ then
7.         $f_{best} = f(\hat{P}^{k'-1}_{[1,m]} \cup \{C_{[m+1,j]}\})$, $\hat{P}^{k'}_{[1,j]} = \hat{P}^{k'-1}_{[1,m]} \cup C_{[m+1,j]}$
8. return $\hat{P}^k = \hat{P}^k_{[1,n]}$
```

Figure 9: DP-RP for Linear Orderings

The speedup arises since we are guaranteed that some cluster begins with index $\pi_1$; thus, for each value of $k'$, we need only record $O(n)$ subsolutions of the form $P^{k'}_{[1,j]}$ instead of $O(n^2)$ optimal subpartitioning solutions. For the special case of a maximum-diameter partitioning discussed above, we achieve an $O(U)$ speedup in Decide_Cluster, again because the *first* module must be $v_{\pi_1}$.

In practice, there will be the question of how to derive a linear ordering from the spacefilling curve. We discuss this question in more detail below.

## 5 Experimental Results

### 5.1 Minimizing Max-Diameter

Our first set of experiments involve the "special case" of DP-RP discussed in Section 3.3, namely, minimizing the maximum cluster diameter, a common objective in the classification literature. We compare DP-RP against three traditional diameter partitioning algorithms: a greedy bottom-up method (AGG) [13], a greedy top-down method (D/Q) [9] and a fast non-hierarchical method (KCenter) [8]. Because of the $\frac{n}{\log n}$ factor speedup obtainable from a maximum diameters objective (Section 4.3), DP-RP for min-max diameter has $O(n^2 \log n)$ time complexity. This is competitive with the $O(n^2)$ complexity of AGG and the $O(n^2 \log n)$ complexity of top-down D/Q. In particular, we find it feasible to consider the entire "tour" generated by the Sierpinski curve instead of only a linear ordering (which would result in an $O(n^2)$ algorithm since Calculate_Cluster_Costs requires at least $O(n^2)$ time when $U = n$).

We ran DP-RP, AGG, D/Q and KCenter on uniformly distributed random pointsets of size 50, 100, 200 and 400 for $k = 2$ through $k = 10$. We considered pointsets chosen from the unit cube in two-, three- and four-dimensional Euclidean space. Table 1 summarizes the improvement obtained by DP-RP over the other three algorithms. The numbers represent averages over 100 iterations for the 50- and 100-point samples, and over 50 iterations for the 200- and 400- point samples. Averaged over partitionings $P^2$ through $P^{10}$, improvements over AGG ranged from -2% to 10% with the amount of improvement increasing substantially with the sample size. DP-RP outperformed KCenter by 5% to 15% though KCenter seemed to improve as the number of dimensions increased and KCenter also can run in $O(n \log k)$ time. Finally, DP-RP was 4% to 9% better than D/Q.

| #Points | AGG | | | KCenter | | | D/Q | | |
|---|---|---|---|---|---|---|---|---|---|
| | 2-d | 3-d | 4-d | 2-d | 3-d | 4-d | 2-d | 3-d | 4-d |
| 50 | 0.932 | 0.976 | 1.022 | 0.845 | 0.895 | 0.943 | 0.910 | 0.917 | 0.977 |
| 100 | 0.917 | 0.951 | 1.001 | 0.842 | 0.895 | 0.940 | 0.933 | 0.906 | 0.972 |
| 200 | 0.909 | 0.936 | 0.977 | 0.833 | 0.890 | 0.939 | 0.944 | 0.915 | 0.964 |
| 400 | 0.903 | 0.927 | 0.967 | 0.818 | 0.895 | 0.939 | 0.951 | 0.943 | 0.955 |

Table 1: Average ratio of DP-RP maximum cluster diameter to AGG, KCenter and D/Q Diameter results for uniformly distributed random pointsets in 2-, 3- and 4-dimensional Euclidean space.

Table 1 does not tell the whole story as the algorithms behave very differently depending on not only dimension and sample size, but also on the value of $k$. Figure 10 shows the average diameter value yielded by the four algorithms on pointsets of size 200 chosen randomly from a uniform distribution in the unit square. We clearly note the inconsistency of D/Q, whose performance clearly depends on the value of $k$. D/Q recursively solves the min-max diameter problem optimally for the largest remaining cluster; hence, D/Q will always yield the best value for $k = 2$, but will generally do badly for $k = 3$ with the resulting 3-way partitioning containing two small clusters and one large cluter. This effect can clearly be seen for powers of 2: when $k$ is a power of 2, D/Q will do very well but for $k$ just less than a power of 2, the partitionings will contain a few large clusters and many small ones, which is generally a poor solution. This is especially obvious for $k = 3$, $k = 7$, and $k = 15$. DP-RP actually has similar behavior since the Sierpinski curve naturally splits into 4 quadrants in the plane. DP-RP also shows better behavior for $k$ a power of two, although the effect is less noticeable and performance is uniformly better than that of D/Q.

One can also deduce from Figure 10 that D/Q generally outperforms AGG for small $k$ but does worse for large $k$; this is expected since D/Q is a top-down algorithm while AGG is bottom-up. Finally, we notice that KCenter has somewhat erratic behavior, e.g., it actually increases cluster diameter when $k = 5$. This occurs because KCenter is based on finding the $k$ "farthest centers"; when $k = 5$, the fifth center will pop up in the middle of the unit square, creating a very large central cluster. We conclude that DP-RP not only outperforms all the other algorithms, but also gives more consistent performance than either D/Q or KCenter.

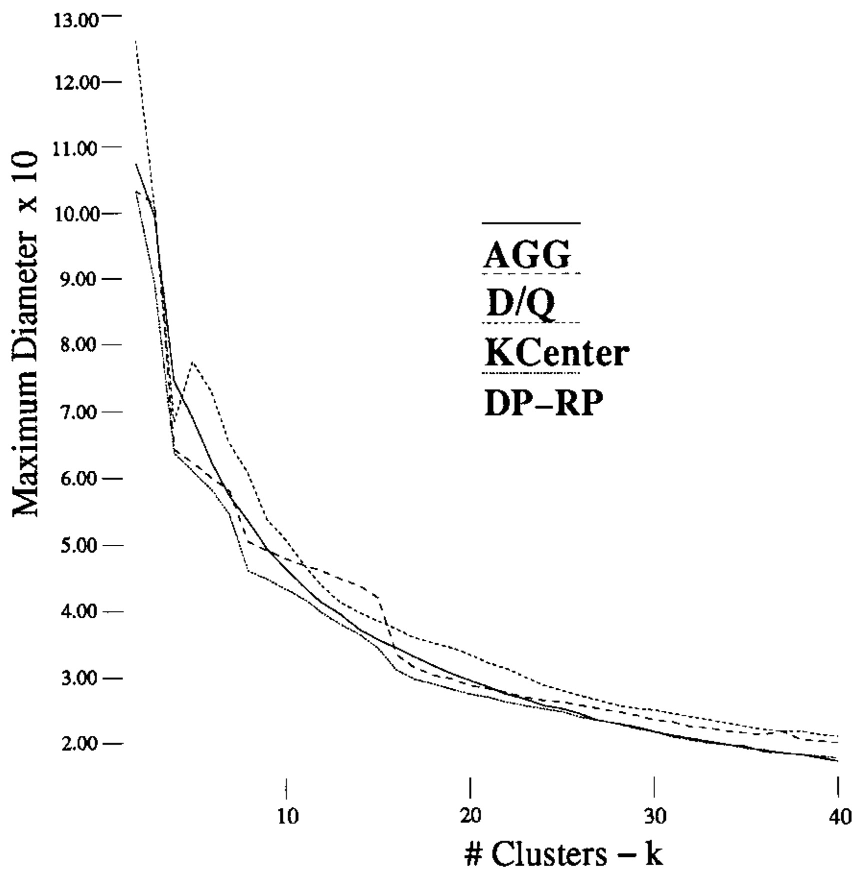

Figure 10: Plot of DP-RP, AGG, KCenter, and D/Q max diameters for uniformly distributed random pointsets in the unit square of size 200. Each point is the average over 50 iterations.

### 5.2 Netlist Partitioning

Our second set of experiments used DP-RP to generate multi-way netlist partitionings according to the Scaled Cost objective. In Section 3, we gave an algorithm which solved the Scaled Cost problem optimally subject to the RP formulation but which required $O(n^3)$ time. This complexity is prohibitively expensive for practical netlist sizes, and so the result we report uses a linear ordering derived from the Sierpinski curve to obtain $O(n^2)$ complexity. We generated this linear ordering from the permutation $\Pi$ by simply removing the largest edge from the associated tour. We found this to be an efficient methodology: after computing the geometric embedding, total runtimes were very short (e.g., 63 seconds for Primary1 and 633 seconds for Primary2 on a Sparc IPX) to generate partitioning solutions for $2 \le k \le 10$.

Since removing the largest edge seems a somewhat crude way of generating the linear ordering, we also generated a subsequent linear ordering based on the split of the 2-way partitioning solution, i.e., if the original solution yielded $P^2 = \{C_{[1,m]}, C_{[m+1,n]}\}$, $\{\pi_{m+1}, \pi_{m+2}, \ldots, \pi_n, \pi_1, \pi_2, \ldots \pi_m\}$ would become the second linear ordering. We repeated this step to generate a third linear ordering and recorded the best partitioning solution obtained by any of the three linear orderings. In practice, this did not yield significant improvements and it may be necessary to derive a better linear ordering construction from the Sierpinski curve, or to even use one of the spacefilling paths shown in Figure 3.

For our experiments, we set $L = 1$ and $U = n$ in order to consider the full range of solutions. For each benchmark, we considered the $d$-dimensional embedding derived from $d$ eigenvectors ($1 \le d \le 10$), and then computed the spacefilling curve for each embedding. This methodology corresponds with the work of [22], which conjectures that $k$ is the correct embedding dimension. Since we considered partitionings from $k = 2$ through $k = 10$, we recorded the best $k$-way partitionings obtained by running DP-RP on 2- through 10-dimensional embeddings. For each benchmark, the best Scaled Cost result is recorded in Table 2.

We compare our method (DP-RP) to KC [1], KP [22], and successive bipartitioning SB [22], which for $k = 2$ is identical to the EIG1 algorithm of [11]. Recall both KC and KP used geometric embeddings of the netlist: KC is a naive diameter-oriented partitioning algorithm, while KP uses more sophisticated partitioning techniques that exploit both geometric and netlist information simultaneously. The Table shows that DP-RP is significantly superior to KP and KC, particularly for $k = 2$ through $k = 5$. For instance, DP-RP averaged 45% improvement over both KP and SB for $k = 2$. We note that DP-RP does poorly as $k$ continues to increase, and this is due to the fact that the restricted partitioning formulation becomes too restrictive for $k$ in this range, i.e., the ordering captures global information but when more local decisions have to be made (i.e., how to further split a 2-, 3- or 4-way partitioning), DP-RP's approach based on a "monolithic" spacefilling curve does not seem to do well. This leaves open future improvement to DP-RP for larger $k$ values.

## 6 Conclusions

We have presented a completely new approach to partitioning which can be applied to a wide range of partitioning objectives. Our approach finds a permutation of points in $d$-space by using the Sierpinski spacefilling curve. We then optimally solve $k$-way partitioning subject to very natural Restricted Partitioning constraints by using dynamic programming. Speedups may be achieved by modifying this general method using a linear ordering or with a different algorithm, as in the case of minimizing the maximum cluster diameter. In practice, DP-RP significantly outperforms the traditional classification algorithms for the max-diameter objective, and also outperforms recent VLSI neltist partitioning algorithms for the Scaled Cost objective.

| Test Case | ALG | Number of Clusters - k (Best dimension) | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 |
| 19ks | DP-RP | 17.6 | 16.8 | 15.6 | 14.3 | 12.7 | 11.7 | 8.37 | 7.74 | 5.44 |
| | KC | | 15.0 | 15.8 | 15.6 | 15.1 | 14.4 | 13.1 | 12.5 | 17.6 |
| bm1 | DP-RP | 24.8 | 22.8 | 20.7 | 18.1 | 14.4 | 11.5 | 8.89 | 6.61 | 5.53 |
| | KC | | 27.6 | 30.6 | 28.6 | 19.8 | 17.9 | 11.1 | 7.0 | 5.8 |
| Prim1 | DP-RP | 38.9 | 36.7 | 35.2 | 31.7 | 28.8 | 26.0 | 22.1 | 14.7 | 13.5 |
| | KP | 44.7 | 41.3 | 32.3 | 33.2 | 31.3 | 29.9 | 21.2 | 14.7 | 13.5 |
| | KC | | 34.6 | 33.6 | 34.4 | 30.7 | 27.5 | 16.4 | 17.4 | 13.5 |
| | SB | 59.4 | 56.2 | 51.0 | 46.6 | 43.2 | 40.3 | 38.9 | 22.5 | 13.5 |
| Prim2 | DP-RP | 13.7 | 13.3 | 12.8 | 12.1 | 11.0 | 9.43 | 7.95 | 6.86 | 5.05 |
| | KP | 15.0 | 15.2 | 13.5 | 11.0 | 10.5 | 10.1 | 9.24 | 7.25 | 4.64 |
| | KC | | 11.7 | 12.0 | 11.8 | 11.5 | 10.4 | 9.0 | 7.5 | 5.9 |
| | SB | 11.1 | 10.6 | 10.4 | 9.56 | 8.44 | 8.47 | 7.56 | 6.56 | 4.78 |
| Test02 | DP-RP | 25.5 | 24.1 | 22.8 | 20.9 | 18.5 | 16.1 | 13.4 | 10.9 | 8.07 |
| | KP | 24.2 | 23.4 | 21.5 | 19.0 | 16.4 | 13.9 | 14.1 | 12.7 | 9.26 |
| | KC | | 21.5 | 21.2 | 21.1 | 21.2 | 23.1 | 23.6 | 19.1 | 30.1 |
| | SB | 25.5 | 23.8 | 21.7 | 20.3 | 18.9 | 14.5 | 13.0 | 11.4 | 8.73 |
| Test03 | DP-RP | 22.6 | 21.1 | 19.2 | 17.1 | 16.2 | 15.2 | 14.3 | 13.0 | 10.2 |
| | KP | 20.6 | 20.1 | 19.8 | 17.6 | 18.0 | 17.5 | 20.2 | 15.0 | 31.2 |
| | KC | | 21.0 | 22.4 | 23.2 | 22.4 | 22.2 | 19.3 | 21.4 | 16.7 |
| | SB | 19.8 | 17.9 | 17.3 | 17.0 | 16.9 | 16.9 | 17.1 | 20.7 | 31.2 |
| Test04 | DP-RP | 22.2 | 19.9 | 17.8 | 17.6 | 16.5 | 15.1 | 12.4 | 8.19 | 5.85 |
| | KP | 17.4 | 17.6 | 20.0 | 15.8 | 15.2 | 14.3 | 14.8 | 18.9 | 66.1 |
| | KC | | 22.1 | 23.8 | 24.4 | 24.3 | 27.2 | 27.4 | 36.0 | 66.1 |
| | SB | 21.9 | 21.6 | 23.3 | 24.4 | 24.9 | 28.2 | 32.3 | 37.4 | 66.1 |
| Test05 | DP-RP | 9.88 | 8.66 | 8.06 | 7.84 | 7.32 | 6.56 | 5.49 | 4.90 | 3.15 |
| | KP | 9.32 | 7.21 | 8.91 | 8.66 | 7.64 | 7.98 | 8.07 | 8.65 | 11.3 |
| | KC | | 11.0 | 10.6 | 10.7 | 11.1 | 10.3 | 8.8 | 10.2 | 10.6 |
| | SB | 9.28 | 84.3 | 6.58 | 6.42 | 6.22 | 6.28 | 6.30 | 6.37 | 8.94 |
| Test06 | DP-RP | 27.1 | 25.1 | 23.7 | 20.2 | 18.4 | 16.6 | 14.2 | 11.3 | 9.2 |
| | KP | 21.3 | 21.6 | 20.7 | 18.5 | 17.2 | 15.0 | 18.0 | 12.1 | 28.6 |
| | KC | | 31.0 | 32.4 | 33.6 | 26.4 | 28.8 | 25.9 | 19.3 | 28.6 |
| | SB | 21.7 | 21.7 | 22.7 | 14.1 | 16.6 | 15.1 | 16.6 | 18.6 | 28.6 |

Table 2: Scaled Cost $\times 10_5$ measures of best $k$-way partitions obtained using $d$-dimensional embeddings, $1 \leq d \leq 10$.

## References

[1] C. J. Alpert and A. B. Kahng, "Geometric Embeddings for Faster and Better Multi-way Netlist Partitioning," *Design Automation Conference* 1993, pp. 743-748.

[2] J. J. Bartholdi, III and L. K. Platzman, "A Fast Heuristic Based on Spacefilling Curves for Minimum-Weight Matching in the Plane," *Information Processing Letters* Vol. 17, No. 4, November 1983.

[3] J. J. Bartholdi, III and L. K. Platzman, "An $O(n \log n)$ Planar Travelling Salesman Heuristic Based on Spacefilling Curves," *Operations Research Letters* Vol. 1, No. 4, September 1982.

[4] J. J. Bartholdi, III and L. K. Platzman, "Heuristics Based on Spacefilling Curves for Combinatorial Problems in Euclidean Space" *Management Sciences* Vol. 34, No. 3, March 1988, pp. 291-305.

[5] D. Bertsimas and M. Gringi, "Worst-Case Examples for the Spacefilling curve heuristic for the Euclidean Traveling Salesman Problem", *Operations Research Letters* Vol. 8, No. 5, Oct. 1989, pp. 241-244.

[6] P. K. Chan, M. D. F. Schlag and J. Zien, "Spectral K-Way Ratio Cut Partitioning and Clustering", *Design Automation Conference* 1993, pp. 749-754.

[7] T. Feder and D. H. Greene, "Optimal Algorithms for Approximate Clustering", in *Proc. 20th Annual ACM Symp. Theory Computing*, 1988, pp. 434-444.

[8] T. F. Gonzalez, "Clustering to Minimize the Maximum Intercluster Distance", in *Theoretical Computer Science*, 38, 1985, pp. 293-306.

[9] A. Guénoche, P. Hansen, and B. Jaumared, "Efficient Algorithms for Divisive Hierarchical Clustering with the Diameter Criterion" in *J. Classification*, Vol. 8, No. 1, 1991, pp. 5-30.

[10] S. W. Hadley, B. L. Mark and A. Vanelli, "An Efficient Eigenvector Approach for Finding Netlist Partitions", in *IEEE Trans. on CAD*, 11(7), July 1992, pp. 885-892.

[11] L. Hagen and A. B. Kahng, "New Spectral Methods for Ratio Cut Partitioning and Clustering", in *IEEE Trans. on CAD* 11(9), Sept. 1992, pp. 1074-1085.

[12] K.M. Hall, "An r-dimensional Quadratic Placement Algorithm", in *Manag. Sci*, 17, 1970, pp.219-229.

[13] S. C. Johnson, "Hierarchical Clustering Schemes", in *Psychometrika* 32(3), 1967, pp.241-254.

[14] R. M. Karp, "Probabilistic Analysis of Partitioning Algorithms for the Traveling-Salesman Problem in the Plane", *Mathematics of Operations Research* 2(3), 1977, pp. 209-224.

[15] E. L. Lawler, J. K. Lenstra, A. Rinnooy-Kan and D. Shmoys, *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization* (Wiley, Chichester, 1985).

[16] N. Megiddo and K. J. Supowit, "On the Complexity of Some Common Geometric Location Problems", in *Siam Journal of Computing*, 13(1), 1984, pp. 182-196.

[17] T. Lengauer, *Combinatorial Algorithms for Integrated Circuit Layout*, Wiley-Teubner, 1990.

[18] F. P. Preparata and M. I. Shamos, *Computational Geometry*, Springer Verlag, New York, NY, 1985.

[19] L.A. Sanchis, "Multiple-way Network Partitioning", in *IEEE Trans. on Computers*, 38, 1989, pp. 62-81.

[20] C. W. Yeh, C. K. Cheng and T. T. Lin, "A General Purpose Multiple Way Partitioning Algorithm", in *Proc. ACM/IEEE Design Automation Conf.*, June 1991, pp. 421-426.

[21] C. W. Yeh, C. K. Cheng and T. T. Lin, "A Probabilistic Multicommodity-Flow Solution to Circuit Clustering Problems", in *Proc. IEEE Intl. Conf. on Computer-Aided Design*, Santa Clara, November 1992, pp. 428-431.

[22] J. Zien, "Spectral K-Way Ratio Cut Graph Partitioning", M.S. Thesis, Computer Engineering Dept., UC Santa Cruz, March 1993.