Polymerization Strategy for the Compression, Segmentation, and Modeling of Volumetric Data

Bruce H. McCormick Brad Busse Zeki Melek John Keyser
Department of Computer Science
Texas A&M University

Abstract:
We present here a new data structure for the representation of volumetric data. The data structure is designed to allow for easy compression, storage, segmentation, and reconstruction of volumetric data. We call our data structure the L-block, abstracting many of the properties of Lego® blocks, and refer to the process of creating and manipulating L-blocks as the polymerization strategy.

The concept of an enhanced volume data set (EVDS) is introduced, where the data set is enhanced by explicitly introducing Boolean labeling of edges between adjacent voxels of the volume data. This enhancement, by "polymerizing" adjacent connected voxels into connected components, facilitates real-time data compression and segmentation of embedded objects within the volume data set. These connected components are packaged in the new container type, the L-block, with the intention of efficiently packaging the connected components with a minimum of adjacent unconnected voxels.

We present the L-block data structure in detail. We describe methods for compressing volume data using the L-block structure, intersecting and merging L-blocks, and segmenting data. The L-block data structure can be viewed as encapsulating a number of other volumetric representation techniques, and we describe this generalization and the costs incurred by the use of L-blocks. While the L-block data structure is general, it was developed to represent scanned brain microstructure at a neuronal level of detail. We highlight the performance of our implementation of the polymerization strategy on a set of sampled neuronal data.

Keywords: Geometric and topological representations, Biomedical applications

1 Introduction

1.1 Motivation

Volumetric representations are needed to model the objects found in volumetric data sets. Sources of such data sets include medical imaging procedures (e.g. MRI) and, more generally, various three-dimensional scanning processes on real-world data.

The work presented in this paper has been particularly motivated by our attempts to scan and reconstruct brain tissue at a neuronal level of detail. The data sets acquired in this work tend to have several distinguishing features. Among them are:

- The full volume data set can be extremely large. The sizes of raw data sets can reach into the terabytes.
- The data of interest within the volume data sets (i.e., the stained neuronal tissue) tends to be sparse, taking up only a modest portion of the overall volume.
- The neurons to be modeled have very long, thin branching structures.
- Data will be acquired at a high rate, and one would like to have a quick way of compressing and storing it in a geometrically meaningful way that facilitates future reconstruction.

We have found current volumetric representation techniques to be deficient in addressing at least one of these features. Due to the potential data size, methods that keep the entire volume in memory at once are unrealistic. Several methods (such as the well-known octree) are poorly suited for modeling long, thin structures. Medial-axis methods, while good for representing neurons, tend to process too slowly and can require too much data to be stored in memory. Pure image and video compression techniques can work well for compression, but fail to give any meaningful insight into the geometric structure of the objects to be modeled. There are a tremendous number of other volumetric data

structures, but a survey of all such structures and their application is not practical here.

1.2 Main Results

We introduce a new data structure designed specifically to address the data set features listed above. The data structure is a general volumetric data structure, and we describe how a number of other structures can be described in terms of it. There are two key components of our data structure.

First, we introduce the concept of an *enhanced volume data set (EVDS)*, where the data set is enhanced by explicitly introducing Boolean labeling of edges between adjacent voxels of the volume data. This enhancement facilitates real-time data compression and segmentation (separating a particular object from its surroundings), as well as subsequent geometric modeling and visualization.

Next, we introduce a new container type, the *L-block*. L-blocks (and coverings with L-blocks) are designed to efficiently package the connected components of the EVDS, with a minimum of adjacent unconnected voxels. We describe a number of basic operations on the L-block structure.

We refer to the process of constructing L-blocks from volumetric data as the *polymerization algorithm.* This is described in detail. We have implemented the L-block structure described, and present the results of its application to some sample neuron data.

1.3 Organization of Paper

The rest of this paper is organized as follows. Section 2 describes the L-block data structure itself. Section 3 describes operations on L-blocks. Section 4 presents the results of an initial application to scanned neuron data, and Section 5 concludes.

2 Representations

In this section we describe the representation of the L-block data structure. We begin by describing the concept of an enhanced volume data set. Next we discuss the L-block data structure itself. Finally, we describe ways in which the L-block data structure can be seen as an abstraction of several other volume representation techniques.

2.1 Domain

We assume that we are given a uniform n-dimensional grid. Every vertex of this grid is assigned a value. The nature of this value may vary. Three possible examples include:

- An integer gray-scale value. This might occur when the data has been obtained from some scanning process, such as from MRI.
- A binary value. This could indicate whether the vertex is or is not in some object. Thresholding or similar techniques might have been used to convert grayscale values to binary values, for example.
- A vector of values. This might arise from multi-spectral scanned data such as a color camera with three channels (RGB).

For simplicity, we will usually refer to a 3-dimensional data set, commonly called a *volume data set*. For 3-D data sets, the voxels form the vertices of the grid. However, the concepts we describe below are equally applicable to other dimensional data sets.

2.2 Enhanced volume data sets

We create an *enhanced volume data set* (EVDS) as described below. The goals of the enhancement are to:

- Allow data compression in real time, in such a manner as to facilitate subsequent segmentation of the volume data set;
- Provide data compression and segmentation strategies that exploit the efficiencies of examining successive serial images, yet are independent of the axis chosen for serial sectioning;
- Separate segmentation clearly from both geometric modeling and visualization of the identified objects in the volume data set.
- Exhibit the statistical basis for the enhancement of the volume data set.

2.2.1 Properties of the EVDS

Given a volume data set, we define an EVDS as follows: in addition to the value assigned to every vertex (voxel) of the grid, selected edges between vertices of the grid are given a Boolean label. *Active* edges are assigned the value 1 and the remaining edges, which are labeled *inactive,* are assigned the value 0.

Edge labeling is used to provide independent information about whether two vertices sharing a common active edge belong to the same underlying object. Boolean labeling {0, 1}, as derived typically from a decision function, is of course a crude estimate of this co-habitation in the same object. Schemes that assign other values to each labeled edge are also possible, but are of less interest for data compression and segmentation. For example, a gray-scale value might be useful for establishing the statistical basis for assigning Boolean values {0,1} to edges of the 3D grid. Or, a vector of Boolean values

might be used to provide separate connectivity for each channel of a multi-spectral scan. We will assume, however, that each edge label is a single Boolean value. The particular decision function used to assign the Boolean values can be arbitrarily complex, and choosing such a function is outside the scope of this paper.

In three dimensions, the data volume is typically created by serially scanning successive *sections* perpendicular to (say) the vertical Z-axis. Here voxels at the same (X, Y) position in two successive images can be conveniently labeled as likely drawn from the same underlying physical object by marking their common vertical edge as active. However, as the specimen could have equally well have been sectioned perpendicular to the X- or Y- axes, we extend the same edge labeling scheme to all three directions.

Any enhanced volume data set (in three dimensions) can have many representations. Most useful for our purposes is an assignment at each vertex of an association {<gray scale value> <edge labels>}, where the Boolean vector <edge labels> indicates the activity level of the edges emanating from the vertex. Vertices at the boundary of the grid (the minimum and maximum X, Y, or Z extents) will have fewer edges. By convention absent edges are labeled 0.

Note that for a regular grid, there may be a choice in the number of edges emanating from any one vertex. We only assume that each vertex has a fixed set of emanating edges, except those absent due to positioning at the boundary of the EVDS. The number of edges emanating from any one vertex is referred to as the connectivity level. For example, consider a regular 3D grid of vertices. If no edges are stored at any vertex (i.e. the data set is not enhanced), we have 0-connectivity. Placing edges in the axis directions (i.e. *(i,j,k)* is connected to *(i+1,j,k)*, *(i,j+1,k)*, and *(i,j,k+1)*).give 3-connectivity. Imagining an axis-aligned cube around the vertex, 3-connectivity would give connections across each face. Connections across the edges as well would yield 9-connectivity, while including the corners in addition would give 13-connectivity. Note that for dimension k, the connectivity level will often be k, as well, and the connectivity level may be at most $(3^k-1)/2$.

A vertex in an EVDS with 3-connectivity can be thought of as having "links" that extend to the neighboring vertices along the three coordinate axes (see Figure 1). Thus it behaves somewhat like a Lego® block, with connections possible in 3 axes.

Figure 1. A vertex in a 3-connected EVDS. The bars show potential links to neighboring vertices.

2.2.2 Notation for an EVDS

For simplicity we restrict our explanation to the 3-connected 3-dimensional case; generalization to other connectivity or dimensions is straightforward. We assume a *3-dimensional curvilinear grid system* has been established. This can be constructed by setting values in a rectangular array of position vectors,

r_{ijk} *(i=0,1,...,I-1; j=0,1,...,J-1; k=0,1,...,K-1)*

and identifying the indices *i, j,*and *k* with the three curvilinear coordinates [4].

Edge vectors, ($e1_{ijk}$ $e2_{ijk}$ $e3_{ijk}$), emanating in the positive direction from a vertex (i,j,k) within the grid are given by

$$e1_{ijk} = r_{i+1jk} - r_{ijk} \text{ for } 0<=i<=I-2$$

$$e2_{ijk} = r_{ij+1k} - r_{ijk} \text{ for } 0<=j<=J-2$$

$$e3_{ijk} = r_{ijk+1} - r_{ijk} \text{ for } 0<=k<=K-2$$

Collectively, these edge vectors enumerate all edges within the three-dimensional curvilinear grid system. Note that for curvilinear coordinate systems having a cyclic coordinate, we can still define edges, e.g.

$$e_{I-1jk}=r_{0jk}-r_{I-1jk}.$$

A single digitized value, g_{ijk}, is assigned to each vertex. An *enhanced* representation of the volume data set retains the gray scale values for each vertex in the grid system. In addition, each edge within the grid is assigned a label of 0 or 1 by a data-dependent edge-labeling function:

$l()$: $l(e1_{ijk}$ $e2_{ijk}$ $e3_{ijk}) = (l1_{ijk}$ $l2_{ijk}$ $l3_{ijk})$

A 1-labeled edge in the grid will be called an *active* edge; a 0-labeled edge will be called an *inactive* edge. The edge label is extended to face-bound vertices of the grid by the convention:

$$l1_{I-1jk} = 0, \quad l2_{iJ-1k} = 0, \quad l3_{ijK-1} = 0$$

In summary, then, a 4-tuple $(g_{ijk}$ $l1_{ijk}$ $l2_{ijk}$ $l3_{ijk})$ is attached to each vertex of the grid.

2.3 L-blocks, Coverings, and Partitions

Given an EVDS, we define L-blocks, L-block coverings, and L-block partitions using that data.

2.3.1 L-blocks

An *L-block* is defined as an n-dimensional rectangular block of enhanced vertex information. The block must be entirely contained within the uniform *k*-dimensional grid of the EVDS. An $(l_1, l_2, \ldots, l_k)$ L-block refers to a block of l_1 vertices in the first dimension, l_2 in the second, etc. Each L-block is defined by its <*header*> information followed by its <*vertex array*>. The <*header*>={<*position*><*template*>}, is given by: (1) the *position*, e.g. *(i,j,k)*, of its least vertex, as indexed within the parent n-dimensional uniform grid, and (2) its *template* $(l_1\ l_2 \ldots l_k)$. Its <*vertex array*> contains the enhanced vertex information (gray scale value(s) and edge labels). In summary the <*packet*> = {<*position*> <*template*> <*vertex array*>} characterizes a L-block.

Given an *k*-dimensional $(l_1, l_2, \ldots, l_k)$ L-block, the number of bits required to store the header is 2*D*, where $D = \sum_{i=1}^{k} D_i$, where D_i is the number of bits needed to store the size of the EVDS (not just the L-block) in each of the *k* directions. For example, a 1024^3 data set would give L-blocks with $D_i = 10$, $D = 30$, and each header requiring at least 60 bits. The vertex array of the L-block requires $(b_r + j)\prod_{i=1}^{k} l_i$ bits, where b_r is the number of bits required to store an individual sample and *j* is the connectivity level. For binary data, b_r would be 1, for grayscale data, b_r is often 8, and for color images, b_r is often 24.

The L-block as a whole can be visualized as a block of vertices, with extensions that demonstrate connectivity. An example is seen in Figure 2.

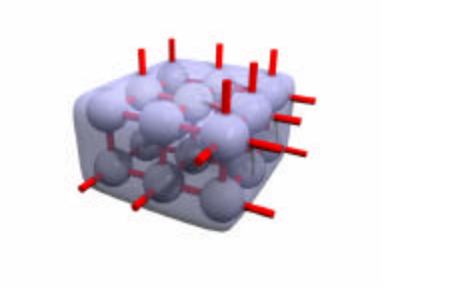

Figure 2. A (3,3,2) L-block. Cylinders represent active edges emanating from the L-block.

2.3.2 Coverings and Partitions

A *covering* $C(\mathrm{A},V) = \bigcup_{a \in A} L_a$ is a covering by L-blocks $\{L_a \mid a \in A\}$ constrained to lie within a volume V, where $C(A,V) \subseteq V \subseteq EVDS$. The L-blocks L_a may overlap and need not be adjacent, and note that the volume *V* may be of arbitrary shape and size (it need not be rectangular). A covering $C(A,V)$ can be given a hierarchical decomposition in terms of other coverings,

$C(A,V) = \bigcup_{i=1}^{m} C(A_i, V_i)$, where

$\{A_i \subseteq A \mid i = 1,\ldots,m\}$ and $V = \bigcup_{i=1}^{m} V_i$.

A *partition*, $P(B,V) = \sum_{b \in \mathrm{B}} L_b$, by L-blocks $\{L_b \mid b \in \mathrm{B}\}$ is a restricted form of covering: $P(\mathrm{B},V) = C(\mathrm{B},V)$, where B labels a partition in V. Similar to coverings, an L-block partition $P(B,V)$ can be given a hierarchical decomposition in terms of other L-block partitions,

$P(B,V) = \sum_{i=1}^{m} P(B_i, V_i)$, where

$\{B_i \cap B_j = \delta_{ij} B_i \mid B_i \subseteq B, i = 1,\ldots,m\}$.

A hierarchical L-block covering (and thus an L-block partition) can be defined by a <*header*> followed by a <*sub-block list*>. The header is identical to that for an L-block, and thus also requires *2D* bits to store. The <*sub-block list*> is composed of a list of pointers to other L-block coverings. The number of bits needed for the sub-block list, then, is $b_v + Nb_m$, where b_v is the number of bits needed to store the maximum number of pointers (note that b_v is never more than *D*), *N* is the number of pointers used, and b_m is the number of bits used for a pointer (usually operating system or compiler dependent). Of course, each covering or L-block that is referenced has its own storage cost. Note that because the sub-block list can point to both L-block coverings and individual L-blocks, it might be necessary to distinguish the two. Often the programming language or compiler will support this type checking; otherwise a single bit can be added to the header to distinguish the two.

An example of a hierarchical L-block covering can be seen in Figure 3.

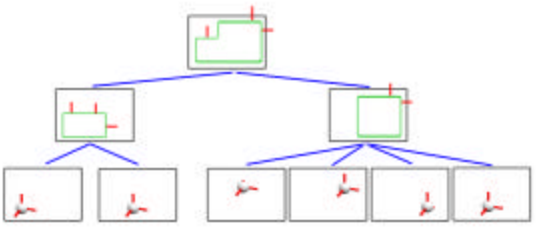

Figure 3. An L-block covering (top of the tree) is formed from the union of two other L-block coverings (middle row). Those L-block coverings are formed from unions of (1,1,1) L-blocks (on the bottom row).

2.3.3 Notation

Hereafter, we will use the following abbreviations. An L-block, consisting of a header and enhanced vertex array will be referred to as an LB, with the size optionally given immediately beforehand. For example, a single 3-D voxel could be described by a (1,1,1)LB. A covering of L-blocks will be referred to as an LBC, and a partitioning of L-blocks as an LBP. It will be assumed that all LBCs and LBPs are expressed hierarchically.

2.4 LBCs as a geometric superstructure

L-block coverings can be viewed as a geometric superstructure, encapsulating several other common volumetric representations. Algorithms that are designed for one of these other formats can be applied just as easily to LBCs. In this way, LBCs can maintain the algorithmic benefits of these other volumetric representations, though possibly with an extra storage cost. We describe below the ways that several common volumetric representations can be represented as LBCs, and the relative additional cost associated with doing so. Note that more information about the various data structures can be found in general computer graphics literature (e.g. [2]).

2.4.1 Unprocessed data

Raw data is often obtained as a set of measured samples taken on a regular grid (e.g. MRI data or the Brain Tissue Scanner data described in section 5). For a k-dimensional space, samples values are given at every point $(c_1, c_2, \ldots, c_k)$, for $c_i = 1 \ldots n_i$. n_i is the number of samples in the i^{th} dimension.

Such data can be directly represented as a single L-block. For example, in three dimensions, an l by m by n block of volume data is stored as an (l,m,n) LB. The header information for the block consists of the position (0,0,0) and the template (l,m,n). For a 0-connectivity LB, the vertex array consists of the measured sample values at each point, and the data is exactly equivalent to the original raw data set. For an enhanced data set, edge labels are stored with each vertex.

The only additional storage cost for the 0-connectivity LB over the raw input data is an additional D (as defined in Section 2.3.1) for the position information. For general connectivity level j, the cost is increased by $j\prod_{i=1}^{n} l_i$. Thus, the additional storage cost for using an L-block instead of the raw data is negligible for a 0-connected L-block, and a bounded multiple (based on connectivity level) of the raw cost for an enhanced L-block.

2.4.2 Enumeration of filled voxels

Another format for storing volume data, useful when the data is sparse, is to simply list the filled voxels. Again, this can easily be stored in the L-block format. Each filled voxel becomes a (1,1,1)LB. These L-blocks are then all joined into a single LBC.

A "normal" enumeration of voxels would take $D+N(D+b_r)$, where D and b_r are defined as in section 2.3.1, and N is the number of filled voxels. The single D factor at the front is needed to store the number of filled voxels, which is at most 2^D. The LBC format will create N (1,1,1)L-blocks, each of which will require $2D+b_r+j$ bits (j is connectivity). Thus, the total storage for the LBC format is $2D+D+N(2D+b_r+j)$ bits, $2D+DN+jN$ more than the "normal" enumeration.

2.4.3 Quadtree/Octree

The octree is one of the best-known methods for compressing 3D volume data, and is related to the quadtree that is used for 2D data. The LBC format can encompass the quadtree/octree format.

Each LBC will include 2^k pointers to other LBCs/LBs. The pointers can be ordered in a standard way, similar to the way they are ordered in a quadtree/octree. A null pointer will indicate that the region corresponding to that pointer is empty. If the pointer is to an L-block, that portion of the region will be completely filled (and the LB will describe a completely filled region). Finally, if the pointer is to another LBC, that LBC will also consist of eight pointers, treated in the same way.

In comparison to a standard quadtree/octree, the additional storage requirements are:

1) An additional $2D$ bits stored with each LBC and LB, describing its position and size. Also, additional bits (2 for quadtrees, 3 for octrees, or D for the general case) are needed at each LBC to store the number of

pointers. Note that all of this is redundant information (and thus not needed in the quadtree/octree representation), but is a part of the LB/LBC structure.

2) A completely filled quadrant/octant of size n will require an additional $(b_r+j)n$ bits to store the fully filled LB. This is a significant overhead. In terms of the overall structure, however, the additional storage for these blocks (not counting the position information mentioned in 1) is exactly $N(b_r+j)$, where N is the number of filled voxels. To compare precisely to an octree/quadtree, b_r would be 1, and j would be 0, making the additional storage bits just the number of filled voxels.

Thus, the additional storage needed for the L-block format is a constant multiple of the number of filled voxels, plus a constant multiple of the number of nodes in the octree.

2.4.4 AABB-tree

Perhaps the most direct use of the LBC structure is to mimic the axis-aligned bounding box (AABB) tree. AABB-trees, commonly used in collision detection, give a hierarchical decomposition of an object by specifying boxes at each level of the hierarchy that are guaranteed to contain all geometry at lower levels. The axes of each box are aligned to the coordinate axes, as opposed to an oriented bounding box tree, where the bounding boxes may have an arbitrary orientation. An interior node in the AABB-tree corresponds to a LBC, with the child nodes corresponding to the sub-blocks of each LBC. The leaf nodes of AABB trees are the primitive geometry (usually triangles). For LBCs, the "leaf" nodes will be the LBs.

There is no inherent increase or decrease in storage requirements between the LBC structure and the AABB-tree structure, except at the leaves. In effect, the two have identical requirements at each node, although the LBC might need to keep track of how many children there are, whereas an AABB-tree is often assumed binary. The storage required at the leaf nodes is a function of the primitive geometry, rather than the data structure.

2.4.5 BSP-tree

Another type of volumetric representation is the binary space partition tree. In this representation, each node of the tree subdivides space into two parts (e.g. in 3D this could be by a plane). The child nodes represent the portion of the object on either side of that subdivision. Leaf nodes are either full or empty, depending on whether the intersection of the halfspaces described by the ancestor nodes is or is not part of the object.

The LBC structure can be used to represent this tree, as well. Interior nodes are LBCs, each with two pointers. Pointers to leaf nodes are either null pointers (indicating an empty region), or point to completely filled L-blocks. The partition that divides the two blocks is implicitly determined by examining the position and templates of the children nodes.

Note that the LBCs can only be partitioned orthogonal to a coordinate axis. A general BSP-tree allows partitioning in any direction. For volume data, this is not an important distinction, as subdivision along some other axis would be poorly defined, and probably not used in a BSP-tree. One other shortcoming of the LBC structure is the inability to represent open (infinite volume) regions, as is possible with the BSP-tree. Again, it is unlikely that this would be needed for representing volumetric data.

The storage requirements for the LBC format of the BSP-tree differ from the standard BSP-tree in the following ways:

1) Each node must maintain position and template information. However, unlike the BSP-tree, it is not necessary to explicitly describe the plane or other structure along which the partitioning occurs. Thus, there is an additional overhead of $2D$ bits for each LBC and LB, but a savings of about $\log_2 k$ bits (to define the axis along which to partition), plus an average of D/k bits (to specify where along the axis to subdivide) at each internal node by not having to store the partitioning information explicitly.

2) Similar to the quadtree/octree representation, leaf nodes incur a significant extra overhead, in that they encode a filled region as an LB, rather than a single bit indicating the region is filled. The additional storage requirement here is identical to that for the LBC describing a quadtree/octree, as given in section 2.4.3.

2.4.6 kD-trees

kD-trees, usually used for point location, are somewhat like a combination of BSP-trees and quadtrees/octrees. Like quadtrees/octrees, each internal node partitions a starting volume of space, while like a BSP-tree, the partitioning is binary and at a specified plane. LBCs can be used to represent kD-trees in exactly the same way they can be used to represent BSP-trees.

The storage requirements for kD-trees are nearly identical to those for BSP-trees as described above. KD-trees can save a small amount of space (the $\log_2 k$ bits mentioned in section 2.4.5) at each level by having a predetermined axis along which to subdivide. The additional cost of an LBC at a leaf is the same as described above for octrees and BSP-trees.

3 Operations

3.1 Binary L-block operations

There are many possible operations that can be defined on L-blocks. We present two binary L-block operations that are quite useful for reconstructing solids from scanned data, as described in section 4. Merging is commonly used when processing data, as seen in section 4. Intersection is important in that it can be used to examine only the portion of a data structure within a certain region of space.

3.1.1 Merge/Union

The merging of two LBs will form another LB. For given input LBs L_a and L_b, assume the header information in dimension *i* is given by position p_i and template value t_i. Then, the new header will have position $p_{i,ab} = \min(\, p_{i,a}, p_{i,b}\,)$, and template index

$$t_{i,ab} = \max(\, p_{i,a} + t_{i,a} - p_{i,ab}, p_{i,b} + t_{i,b} - p_{i,ab}\,)$$

It is necessary to assign values to all the vertices within the merged L-block. For those corresponding to vertices from the input LBs, this is straightforward. For the other vertices, no information is known, and it is necessary to assign "empty" values with no active edges to each of these vertices.

Considering LBCs, the merging of two LBCs is a straightforward process. Either a new LBC is created, with pointers to the input LBCs as sub-blocks, or the pointers of the two input LBCs are merged into a single list. In either case, the header information is adjusted as for merging two LBs. LBCs are closed under union.

3.1.2 Intersection

The intersection of two LBs, L_a and L_b, $L_{ab} = L_a \cap L_b$, is either an LB or the empty set $\varnothing$. For LBPs this representation is unique: Let A and B be partitions with $(a, m \in \mathrm{A} \mid b, n \in \mathrm{B})$, then $L_{ab} = L_{mn}$ implies $a = m, b = n$.

Lemma: The class of coverings with respect to a volume V is closed under intersection.

The intersection of two LBCs, $C(\Delta, V) = C(A, V) \cap C(B, V)$, is again an LBC. The resulting LBC is formed from a collection of LBs: $\{L_{ab} = L_a \cap L_b \mid a \in A, b \in B, a\, b \in \Delta\}$. In general the covering set Δ will have members in neither A nor B.

Lemma. The class of partitions with respect to a volume V is closed under intersection.

The intersection of two LBPs, $P(\Delta, V) = P(A, V) \cap P(B, V)$, is again a partition formed as a collection of LBs in the same way as for coverings. Here Δ is a partition, as the L_{ab} are disjoint:

$$L_{ab} \cap L_{mn} = (L_a \cap L_b) \cap (L_m \cap L_n)$$
$$= (L_a \cap L_m) \cap (L_b \cap L_n)$$
$$= d_{am} L_a \cap d_{bn} L_b = d_{am} d_{bn} L_{ab}$$

That is, the intersection of the two partitions can be expressed as a collection of L-blocks.

3.2 The polymerization segmentation strategy

Our intention is to identify objects of interest from within a volumetric data set. To this end the *polymerization segmentation strategy*, formalized below, views active edges within the EVDS as hardening (polymerizing) into a structure not unlike a jungle gym and the objects of interest lifted (segmented) out of the 3D block-structured grid. This strategy will be successful to the extent that the data-dependent edge-labeling function $l_{ijk}(\ )$ captures the connectivity of the underlying physical objects in the scanned block. To bound the size of the connected component, we have defined coverings and partitions with respect to a volume V. Several definitions and lemmas are introduced below to help formulate the polymerization process.

3.2.1 Volume and regions of interest

A *volume of interest*, VOI_A, is a covering $C(\mathrm{A}, V)$ of a connected component G, where $G \subseteq C(\mathrm{A}, V)$. A *volume segmentation strategy* then consists of identifying volumes of interest, $\{VOI_1, VOI_2, ..., VOI_n\}$, that collectively provide a covering of all connected components within the volume V.

For the special case where the volume V is a planar slice of the EVDS, a *region of interest*, ROI_A, is a covering $C(\mathrm{A},V)$ of a connected component G, where $G \subseteq C(\mathrm{A},V)$. An *image segmentation strategy* then consists of identifying regions of interest, $\{ROI_1, ROI_2,..., ROI_n\}$, that collectively provide a covering of all connected components within the volume (image) V. ROIs from successive images can then be threaded together by their active vertices to form VOIs.

3.2.2 Graph of a covering (dag of a partition)

Two distinct L-blocks, L_a and L_b, are *joined* if their vertices share at least one active edge. Adjacent LBs may or may not be joined.

We define the graph $\mathbf{G}$ of a covering $C(A,V)$, as follows:

1) *Vertices of the graph* $\mathbf{G}$: Each L-block in $C(A,V)$ is assigned a vertex in $\mathbf{G}$.

2) *Edges of the graph* $\mathbf{G}$: An edge $e_{ab} \in \mathbf{G}$ links two L-blocks iff the L-blocks are joined. In general, the edge is undirected.

We define the dag of a partition $P(A,V)$ similarly, except all edges of the graph are directed, assigned the direction consistent with all vertices sharing an active edge. The dag of a covering or partition may contain more than one connected component.

Lemma. Every connected component within a covering $C(A,V)$ resides in a connected component of the graph of the covering. The converse is not in general true.

3.2.3 Cost of a covering or partition

Useful segmentations, by coverings or partitions, of the active vertices in a volume V, where $V \subseteq EVDS$, must be efficient: avoiding overly large blocks (covering excessive white space) or requiring large numbers of small blocks. The quality of the covering (partition) can be controlled in part by assigning a cost to a covering (partition). For a covering $C(A,V)$, we assign a cost $\$(C(A,V)) = k + \sum_{a \in \mathrm{A}} \$(L_a)$, with $\$(L_a) = l + N_a m$, where k is a cost associated with the covering (irrespective of the L-blocks), l is a cost associated with a single L-block, m is a cost associated with a single vertex, and N_a is the number of vertices in L_a. An identical cost formula will be used for partitions.

Although many such cost functions are possible, one natural one is based on the memory requirements of the structure, as outlined in the bit costs described in section 2.3. In this case, the parameters k and l represent the cost of the header information, while m represents the cost of storing the grayscale value and edge information at each vertex.

The objective, then, is to find a minimal cost covering for the given volume. Notice that for $l = 0$, any covering of all active vertices by L-blocks with (1 1 1)templates is of minimal cost, provided the (1 1 1) template is permitted in the covering. For $l > m$, minimal coverings will trade off using fewer blocks with covering more white space. As l increases, a minimal cost covering will, in general, use fewer blocks of larger size. Counter to intuition, for any covering $C(A,V)$ and cost parameter $0 < l$, a partition $P(A',V)$ of lower cost may not exist.

3.2.4 Compressed representations of an EVDS

Connected components in the extended volume data set are of particular interest, as these are the substrata upon which objects in the volume data set are modeled. Focusing on its connected components, and efficiently packaging these within LBCs, can significantly compress an EVDS. Given an EVDS, we use the polymerization strategy to compress the data, retaining only what is needed.

Isolated vertices, those having no active edges, occur regularly in scanned volume data, often due to "noise." Such vertices can be ignored, or at worst, packaged in small LBs for separate consideration, should their gray scale value exceed some threshold. In this situation the remaining vertices outside the coverings can be treated as "white space", and ignored in subsequent image processing. The content of the EVDS, exclusive of its "white space", is then captured in the LBC (or LBP). The packets of the covering can be stored in any order, and can be transmitted without concern for their order of arrival at the receiving site.

At other times only the boundaries of objects warrant consideration. Here "black space" LBs, whose every

associated edge is active, can be separately noted, and suppressed.

Volume data generated by serial sectioning and scanning of a three-dimensional specimen can be compressed in real time by incrementally generating the EVDS. As each consecutive image is scanned, only its immediate predecessor need be retained in memory while the current image data is enhanced and incrementally added to the evolving EVDS. For example, let consecutive serial sections be scanned in the XY plane at depths of Z and Z+1 respectively. The Z+1-plane image data is used to enhance the Z-plane image data. *Regions-of-interest (ROIs)* in the Z-plane image are then packaged in (m n 1) L-blocks, stored as packets, and added to the evolving compressed representation of the EVDS. Black space blocks can be deleted at this time or their processing deferred.

4 Application

We have implemented the L-block data structure as described, and have applied the polymerization strategy to a sample data set of scanned neuronal data. In this section we describe the results of our work in order to demonstrate the utility of our approach for the compression, storage, segmentation, and reconstruction of volume data.

4.1 Forming the EVDS

To test our implementation, we use a volume data set obtained from a set of 14 serial scanned images. The images were obtained from Golgi-stained mouse brain tissue imaged through a light microscope with incoherent illumination. Registration of the images along the vertical axis was done by hand, and for this reason the regions near the borders of the images are unreliable (i.e. the areas around the border of one image might contain data at positions unavailable in the images above and below). Each voxel of the data set represents a volume of 0.37 µm by 0.37 µm by 0.5 µm. Note that we will consider the Z direction to be the axis perpendicular to the images, and we will sometimes refer to individual images as "sections." The details of this data set have been described in [1]. A sample section is shown in

Figure 4.

Eventually, we will use our L-block structure to process data obtained from the Brain Tissue Scanner (BTS) [3]. The BTS is a unique instrument developed at Texas A&M that uses a diamond knife

to concurrently cut tissue and scan the tissue at the knife edge. Although the BTS data will be over a larger volume, arrive at a faster rate, and use different (coherent and more uniform) illumination, we anticipate it will have similar characteristics to the sample data, for Golgi stains. Note, however, that other staining techniques may have different properties.

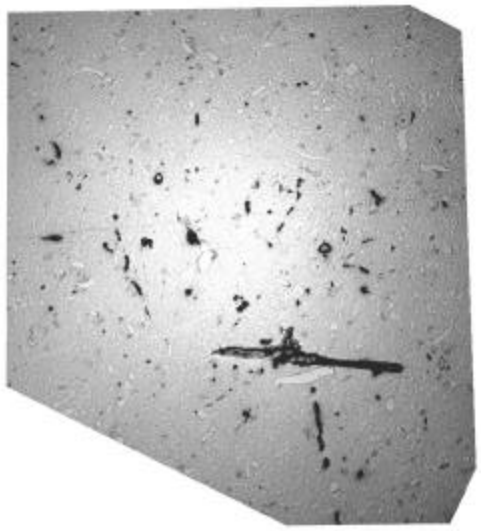

Figure 4. One of the sections in our dataset. The large black region is a stain smear.

For the examples presented here, we use very simple functions to determine valid vertices and edge labels. We consider vertices valid if they pass a simple thresholding test (i.e. have grayscale values above a certain level and below another level). Edges are labeled active iff both of the adjacent vertices are valid. While future reconstruction efforts will likely involve more complex labeling functions, these suffice for making an EVDS for initial testing purposes.

4.2 Compression of Data

The memory needed to hold useful amounts of uncompressed neural data is exceedingly large. For example, the raw BTS data for an entire mouse brain will be approximately 29 terabytes. For this reason, space saving features of the L-block structure and compression of the initial data are very important.

The majority of data compression takes place during the thresholding stage. Voxels that do not pass the thresholding stage are considered "white space," and it is assumed that they can be ignored thereafter. The EVDS is partitioned into 2 by 2 by 2 cells. If any of the voxels in a cell is valid (i.e. passes the threshold), that cell is stored as a (2 2 2)LB. The compression achieved will depend on the stain, the threshold used, and the density of the data. For our sample Golgi-stained data set the initial data set requires

approximately 112 MB of storage space. With realistic threshold levels, we form 47258 LBs requiring about 4.4 MB to store, yielding a compression factor of approximately 25.

We then provide additional compression by combining LBs where appropriate. Merging L-blocks has the advantage of eliminating the overhead of the header information. While combining two (2 2 2)LBs into, say a (4 2 2)LB is straightforward, combining larger LBs with smaller ones may be more problematic. Because LBs store all data in a rectangular volume, expanding an LB might require storing "white" space along with relevant data. To determine whether or not it is appropriate to create such LBs, we use a cost function based strictly on the relative storage requirements for the merged and unmerged LBs. We consider merging LBs in each of the positive X, Y, and Z axis directions. The L-blocks are extended if the space that would be saved by eliminating L-block overhead is greater than the space lost by storing empty data. Figure 6 shows a close-up image of merged LBs (drawn as wireframe boxes) from the sample neuron data set. Figure 7 shows the merged LBs for the entire sample data set, and Figure 8 shows the merged LBs for a portion of the data set, overlaid with the valid data. For the entire data set, our merging strategy reduces the total number of LBs to 12841, requiring less than 3.7 MB of storage.

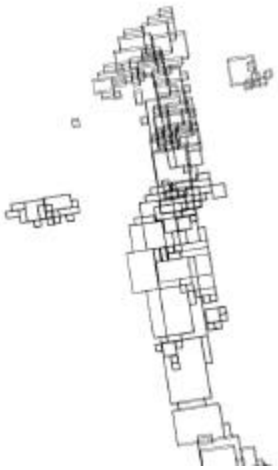

Figure 6. Merged L-blocks.

Noise reduction can also be used to reduce data storage. LBs that have no active edges emanating can be eliminated. Such LBs are unlikely to be a part of a neuronal structure, and are most likely due to noise in the input data. For our sample data set, noise reduction reduces the total number of LBs to 7525, requiring only 2.3 MB to store. Together, noise reduction and LB merging provide a factor of 2x compression in our sample data, giving an overall compression of approximately 50x.

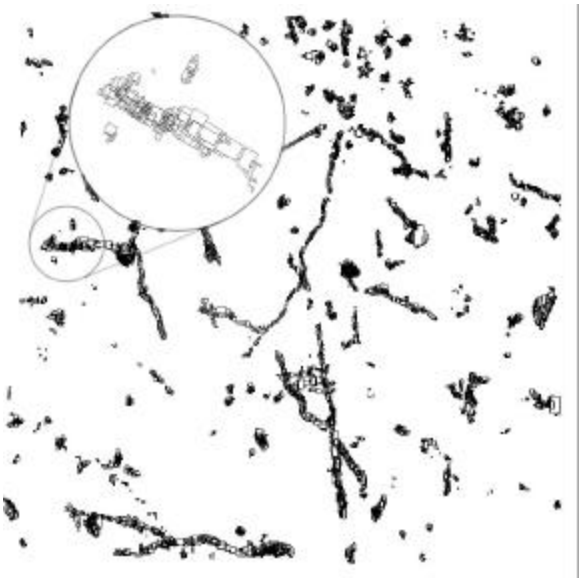

Figure 7. The merged L-blocks formed for the entire data set. The portion used to form Figure 6 is highlighted.

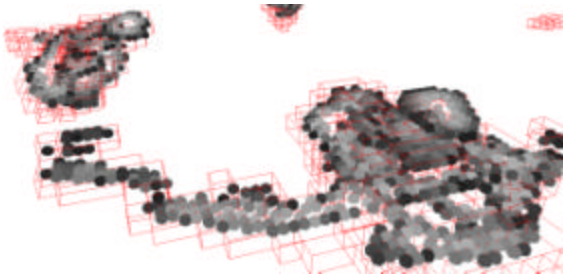

Figure 8. A portion of the reconstructed data showing the valid data within the L-blocks.

Note that these strategies are well suited for processing BTS-style data. BTS data will arrive one "section" at a time, and each section must be processed in real time. Due to the amount of data, it is not practical to store many sectional images in memory at once. Forming the EVDS generally requires only two sections, though some edge labeling functions may require a small, fixed number more. Forming (2 2 2)LBs likewise requires only two sections (though a third might need to be kept in memory for edge labeling purposes). Merging LBs requires only storage of the (possibly already combined) LBs that cover portions of the immediately preceding section – typically there will be only a few such LBs, and in any case, the number is bounded by the size of the section. Although this process biases LB merging in the Z direction, it is necessary due to time and memory constraints.

Finally, noise reduction can be applied to only those LBs currently in memory.

4.3 Data segmentation

Taking advantage of the fact that neural data (Golgi stained) is both sparse and clustered, our data is further combined into clusters, expressed as an LBC. Clusters are defined as groups of interconnecting L-blocks. If two L-blocks border on each other and at least one of the voxels composing that border has an active link to a voxel in the other L-block, both L-blocks are considered to be in the same cluster. Since the voxels themselves are used to determine cluster boundaries, this scheme effectively segments the data, i.e. it does not group two pieces of data that should have been separate. Notice that it is possible for two different LBCs to have L-blocks that overlap in area, but relevant data in one L-block will be empty space in the other, so no harm is done. Also, if LBs are clustered before merging, the space of LBs to be examined for potential merging is reduced, thus speeding up the algorithm.

Figure 9 shows an example of the connectivity between the LBs of Figure 6. The lines in the figure indicate that the LBs centered at each endpoint are joined by an active edge, and thus are grouped together in a cluster.

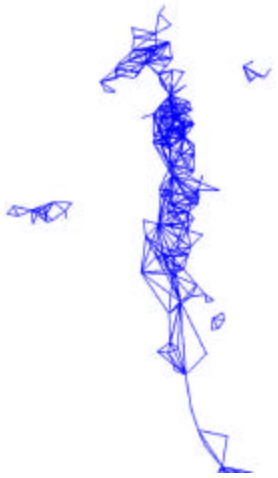

Figure 9. The connectivity graph for Figure 6. LBs in the same connected component are grouped in an LBC.

Noise reduction features are also implemented at the cluster level to conserve space. Two types of noise are targeted in our implementation. The first type is clusters that are too small to be a valid neural structure by themselves and too isolated to be a fragment of a larger structure. The second type is stain smears – medium to large swaths of data that exist entirely on the XY plane (arising from the staining agent smearing as the tissue was being cut).

Figure 4 includes a stain smear. The rules used to identify these noise clusters are ad hoc, and though very effective on the sample data sets, would need to be altered for other data sets.

Note that this segmentation can be performed locally. That is, LBCs can be formed based on just a limited amount of data in memory at one time. Again, this is necessary due to the potentially large amount of data.

For the sample data set, the number of clusters formed from the data after initial thresholding is 5610. Section 4.2 describes the exact number of LBs and the total memory requirements. After noise reduction is used to eliminate some clusters entirely, the total number of clusters is reduced to only 1654.

4.4 Neuron Reconstruction

With the construction of the LBCs, the neuron data is effectively segmented. This, however, is not sufficient information to effectively reconstruct a neuron. Due to staining irregularities, noise, etc., the LBCs constructed during segmentation are unlikely to capture the complete neuron geometry. In fact, the particular connections between LBs are expressed only via the active edges – the LBC does not encode such information. We have implemented a method for extracting a neuron model based on the segmented LBC. This is presented primarily to show the utility of the LB/LBC data structure, and not as an ideal neuron reconstruction algorithm. Our approach makes use of both the hierarchical LBC structure as well as the structure of the individual LBs to reconstruct the neuron.

We begin by dilating the L-blocks within each cluster (i.e. expanding the apparent size of the L-blocks by adjusting the header information). After doing this, LBs will begin to overlap (in extent) "nearby" LBs. We join the LBCs that overlap when dilated, effectively filling in the gaps that could be missing due to noise. We form an expanded connectivity graph, linking the LBs that overlap when dilated. This can be visualized as in Figure 9. Of course, this also joins some LBCs that should be kept separate. We must therefore reduce the complexity of the connectivity graph in order to identify the major threads along which the neuron lies.

The expanded connectivity graph is then simplified into a tree format (i.e. a hierarchical LBC) that captures the major dendritic threads passing through the sample section set. This is done by first temporarily removing "fine-scale" detail, which can be identified based in part on the LB sizes.

Removing this detail simplifies the connectivity graph considerably, after which we apply graph algorithms to simplify the graph further. In the end, a "thread axis" (possibly including branching) is constructed around which the hierarchical LBC can be created. Given the hierarchical LBC, a medial axis approximation can be obtained. Using radius estimation, the medial axis representation can be iteratively refined to match the L-block representation. A picture of the thread axis so obtained is shown in Figure 10. A 3D reconstructed view can be seen in Figure 12. Note that because the reconstruction region is so small, we have only portions of each neuron, and thus we do not capture much of the branching structure typical of neurons.

Figure 10. The major thread for Figure 6 data.

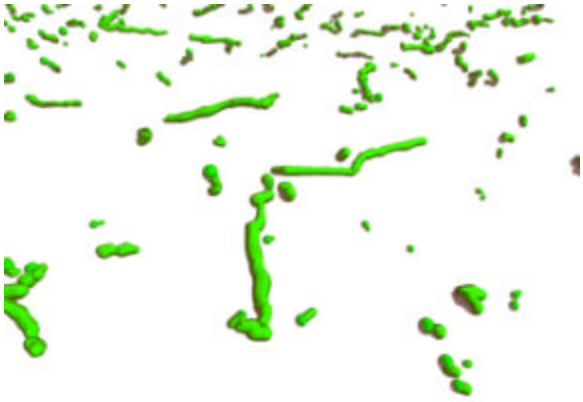

Figure 12. A view of the reconstructed neurons from the data set.

5 Conclusion

5.1 Summary

We have presented a data structure for the representation of volumetric solids. The data structure is general, and can be seen as a superstructure that encompasses many other common volumetric data structures. We have described the basic operations on this structure, including the polymerization strategy for segmenting data from a volumetric data set. We have outlined how this strategy can be used for the compression, storage, segmentation, and reconstruction of volume data. The L-block data structure and polymerization strategy have been implemented, and we have demonstrated that it can be used effectively in the reconstruction of neuron data.

5.2 Features of Our Data Structure

To conclude, we point out a number of features of our data structure that give it an advantage over other structures, in certain circumstances.

- The structure can be used for data compression and segmentation in an incremental manner. That is, only a limited amount of the data set needs to be in memory at any one time, making it useful for application in a real-time scanning environment.
- The compressed data has clear geometric meaning, as opposed to compression methods such as standard image compression (e.g. JPEG/MPEG), where most geometric meaning is lost.
- The data structure is well-suited for describing long, thin objects, as well as branching structures. Many other methods (e.g. octrees) are geared more toward compact, fatter objects.
- The structure is well suited for sparse, clustered data, for which it provides very good data compression and segmentation.
- The data structure is very general, capable of mimicking the operation of other data structures. Thus it is flexible and can be applied easily to a wide variety of situations.

5.3 Future Work

Among the directions for further work are:

- A number of operations on L-blocks and their coverings could be defined. Though many operations are straightforward, some (such as reordering the hierarchy of a LBC) are quite challenging.
- The polymerization strategy on volume data is highly dependent on the edge labeling strategy. Developing an effective edge-labeling strategy is thus important for the polymerization strategy to be effective.

Acknowledgement

This work was supported in part by Texas Higher Education Coordinating Board ATP grant 000512-0146-2001.

References

[1] Burton, B.P., B.H. McCormick, R. Torp, and J.H. Fallon. Three-dimensional reconstruction of neuronal forests, *Neurocomputing*, 38-40:1643-1650, 2001.

[2] Foley, J. D., A. van Dam, S. K. Feiner, J. F. Hughes. *Computer Graphics Principles and Practice, Second Edition in C*. Addison-Wesley, Boston, 1996, pp. 548-557.

[3] McCormick, B.H., *Development of the Brain Tissue Scanner*, Technical Report, Department of Computer Science, Texas A&M University, College Station, TX, March 18, 2002. Available from http://research.cs.tamu.edu/bnl

[4] Thompson, J.F., B.K. Soni, and N.P. Weatherill, *Handbook of Grid Generation*, CRC Press, 1999, pp. 1-4.