Efficient Generation of Contour Trees in Three Dimensions

by

Hamish Carr

B.Sc.(Hons.) 1987, LL.B. 1990, B.C.Sc.(Hons.) 1998 University of Manitoba

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

THE FACULTY OF GRADUATE STUDIES

(Department of Computer Science)

we accept this thesis as conforming
to the required standard

The University of British Columbia

April 2000

Abstract

Many scientific fields generate data in three-dimensional space. These fields include fluid dynamics, medical imaging, and X-ray crystallography.

In all contexts, a common difficulty exists: how best to represent the data visually and analytically. One approach involves generating level sets: two-dimensional surfaces consisting of all points with a given value in the space. With large datasets common, efficient generation of these level sets is critical. Several methods exist: one such is the *contour tree* approach used by van Kreveld et al. [26].

This thesis extends the results of van Kreveld et al. [26] and Tarasov & Vyalyi [23]. An efficient algorithm for generating contour trees in any number of dimensions is presented, followed by details of an implementation in three dimensions.

Contents

List of Figures

Chapter 1

Introduction

1.1 Overview

Applications in several disciplines require sampling of some physical quantity in three dimensions, followed by visualization of the data thus acquired. These disciplines include medical imaging [15, 18, 17], fluid dynamics [17], and X-ray crystallography [13, 6].

The sampling process produces a finite number of data points with associated values. Most, if not all, visualization techniques assume an underlying real-valued function, and approximate it by means of some interpolation function $f(x)$ over the volume of space under consideration. I defer definition of this interpolation function to Def. 2.4, p.10.

Many techniques for visualizing such data exist, the technique used in this thesis is that of *level sets*[1]: surfaces defined by $\{x \in \mathbb{R}^3 : f(x) = h\}$ for

[1]Often called isosurfaces: see discussion under Def. 2.6, p.11

some fixed value h.

Using X-ray crystallography as a driving problem, I discuss existing techniques for generating level sets, in particular techniques based on a data structure called the *contour tree*. I then propose an efficient and practical algorithm for constructing contour trees in arbitrary dimensions. After analyzing the performance of this algorithm, I consider implementation issues, and present some experimental results.

1.2 X-ray Crystallography

One of the major problems in biochemistry is the analysis of protein structure and conformation. It is known that the function of a protein is intimately connected to it's conformation [21], which is determined by the spatial locations of the atoms in the protein, and the interaction of the electron clouds surrounding each atom.

Techniques exist to determine the sequence of amino-acid residues in a protein [21]. This sequence defines the number and connectivity of the atoms in the protein, but not the spatial location of the atoms.

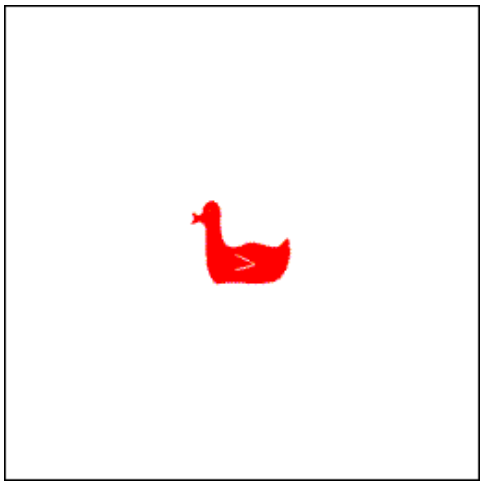
X-ray crystallography is concerned with determining the locations of the atoms in the protein. Although minor variations exist, the following is an overview of the experimental procedure [9].

A quantity of the protein in question is crystallised: it is assumed that this lines up many copies of the protein in similar orientations. The resultant crystal is then bombarded with X-rays. The X-rays interact with the electrons

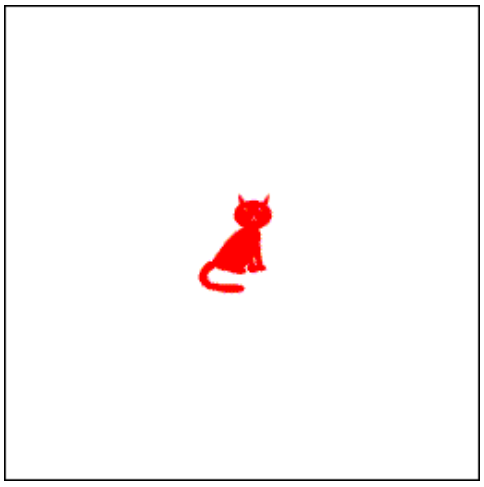
associated with each atom, and are sometimes refracted. These refractions are measured by a planar array of detectors. The process is repeated with the detectors at varying distances from the crystal: this produces a three-dimensional dataset, consisting of scalar values (the measured intensities at the array).

These measurements, however, do not correspond directly to the electron distribution in the crystal, but are related to the Fourier transform of the electron distribution. If the electron distribution in a typical molecule is known, the Fourier transform of the distribution can be computed: this is defined over complex numbers. Since the experimental data is scalar-valued rather than complex-valued, the Fourier transform is underdetermined. It is, however, known that the data corresponds to the amplitude of the Fourier transform: only the phase information is lost [21].

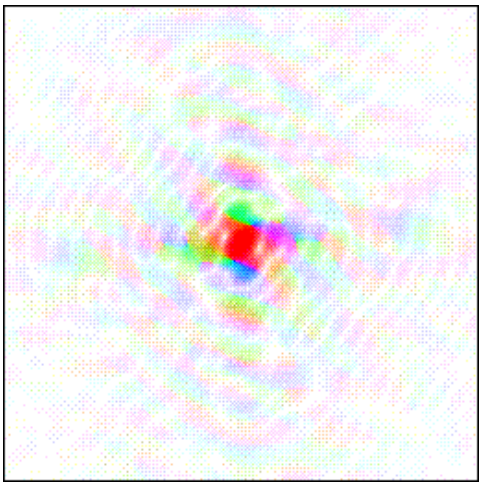
The locations of the atoms are then found by an iterative procedure. The researcher hypothesizes the missing phase information, and performs the inverse Fourier transform to recover spatial data, in the form of an *electron density map*. The researcher then visualizes the electron density map, and forms further hypotheses about the locations of the atoms, and about the missing phase information. This process is repeated, refining the locations of the atoms at each iteration, until the researcher is reasonably certain that the locations of the atoms have been determined. It is possible, however, to make mistakes in this procedure, and recover incorrect data. In particular, the hypothesized phase information can lead to an entirely incorrect map.



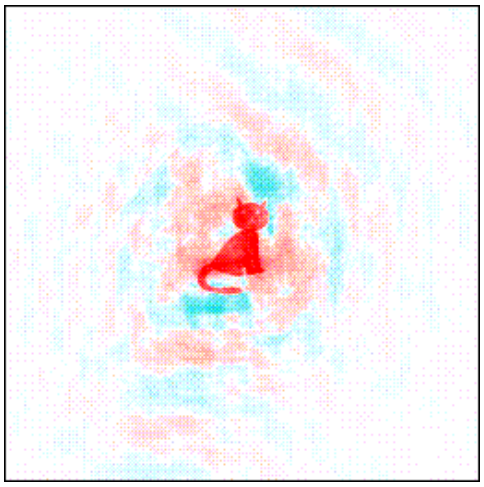
(a) A duck



(b) A cat



(c) Cat Phase, Duck Amplitude in Reciprocal Space



(d) Cat Phase, Duck Amplitude in Real Space

Figure 1.1: Errors in Hypothesizing Phase Information

As an example, consider Fig. 1.1(c), from [7], which combines the magnitude information from the duck, and the phase information from the cat. The result of this is the image (Fig. 1.1(d)), in which the the cat (phase) information dominates visually.

It is a common and useful assumption that the nuclei of atoms are local maxima (at the centre of an electron cloud), and that molecular bonds are *saddle points*: bridges between two electron clouds [9]. For this reason, visualization techniques which aid in the location of local maxima and saddle points are preferable. In particular, note that local maxima can be located in principle by choosing a high *isovalue* (Def. 2.5, p.11) to generate *contours* (Def. 2.6, p.11). If the isovalue is chosen appropriately, this creates a small contour around each local maximum. As the isovalue decreases, these level sets will gradually merge at saddle points, allowing us to detect the saddle points.

However, a major problem exists: not all local maxima are equally high-valued, and some are lower-valued than the saddle points between the higher maxima. When this happens, it is impossible to select a suitable isovalue so that one can correctly identify all maxima. Either the lower-valued maxima will fail to appear, or the isovalue chosen to show the lower-valued maxima will be lower than the high-valued saddle points, causing the higher-valued maxima to blur into each other.

Two possible causes of this problem exist: thermal noise, and type of atom. Each protein is composed of a *backbone* or *main chain* of carbon atoms:

the locations of these atoms are generally highly constrained, producing high-valued maxima. Atoms off the main chain are somewhat more flexible, and have freedom to rotate about one or more of the bonds in the side chain. As a result of thermal noise, the maxima for these atoms will be diffused, and will be lower and flatter than for atoms on the backbone. Similarly, there is some variation between the heights of the maxima corresponding to atoms of different elements.

One solution to this problem is local contouring: generating contours for different isovalues in different regions of the data. See Sec. 9.5, p.98 for further details.

1.3 Desiderata

From the description above, we extract a number of desiderata for visualizing crystallographic data:

1. Local maxima and saddle points should be easily located. As noted above, level set techniques can be used to do this.

2. Level sets should be able to be generated efficiently, even for large datasets, to facilitate changing the isovalue at interactive rates.

3. It should be possible to compute and display contours at different isovalues in different portions of the image.

1.4 Assumptions about Data

I make certain assumptions about the data: most of these are necessary for the contour tree approach (Assns. 1.4, 1.3, 1.5). The remaining assumption simplifies the implementation (Assn. 1.2).

Definition 1.1 *The input data is defined to be a set of real-valued measurements in a bounded volume V in $\mathbb{R}^d$:*

$$\mathcal{P} = \{(x_i, h_i)\} : x_i \in V \subseteq \mathbb{R}^d, h_i \in \mathbb{R}$$

Assumption 1.2 *Regular Rectilinear Data*

I assume that electron density maps are datasets sampled at regular intervals on a three-dimensional rectilinear grid.

Assumption 1.3 *Simplicial Mesh*

I assume that the mesh on which the interpolation function is based is a simplicial mesh (Def. 2.3, p.10) . This assumption is necessary for the contour tree approach, but not necessary for other level set generation techniques. Methods for ensuring that the mesh is simplicial are discussed in Sec. 9.1, p.91.

Assumption 1.4 . *Piecewise-Linear Interpolation Function*

I assume that the interpolation function (Def. 2.4, p.10) is piecewise-linear. Although this assumption is not strictly necessary, it greatly simplifies some preliminaries (for example, Lemma 5.21, p.69). It is also is the most

common interpolation function (see Def. 2.4, p.10). When combined with Assn. 1.3, this assumption means that the interpolation function inside a given simplex will be the linear interpolation between the vertices of the simplex.

Assumption 1.5 *Uniqueness of Data Values*

I assume that no two values in the data are identical. Without this assumption, it is not guaranteed that there will be a unique contour tree for a given set. The solution adopted, "simulation of simplicity" [11], is discussed in Sec. 9.3, p.97.

This assumption is also required to guarantee that the critical points occur only at vertices of the mesh (Sec. 4.1.1, p.26), and applies to all contour tree-based methods.

Chapter 2

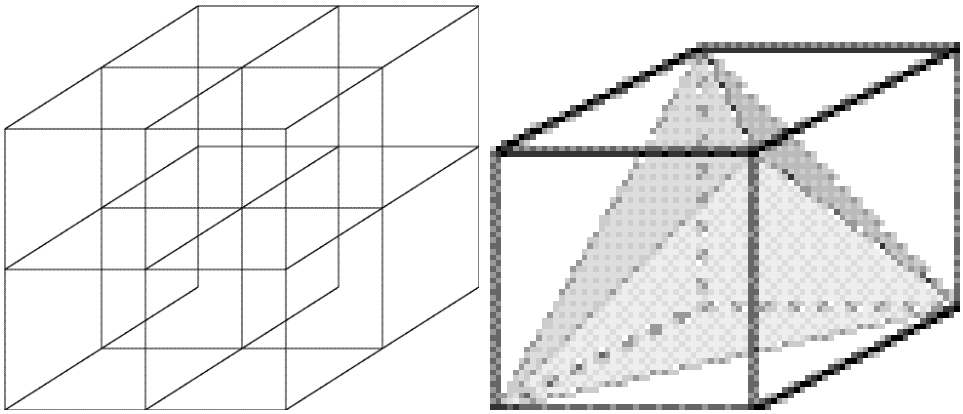
Definitions

In order to simplify discussion, some relevant terms are defined. Since this thesis is printed on two-dimensional paper, illustrations of two-dimensional datasets will sometimes be used for clarity.

Definition 2.1 *A* mesh *in* $\mathbb{R}^3$ *is a collection of polyhedral volumes or* cells *that completely fill the volume* V*. The vertices, edges, and faces are shared between adjacent polyhedra, and the vertex set is the set of data points* $\{x_i\}$*. Note that definitions of* mesh *used elsewhere may differ slightly from this definition.*

In practice, meshes are frequently rectilinear meshes (Def. 2.2), hexahedral meshes (which are similar to rectilinear meshes) or simplicial meshes (Def. 2.3).

Definition 2.2 *A* rectilinear mesh *is a mesh (Def. 2.1) in which each cell is a rectangular prism, or* voxel*, as in Fig. 2.1(a).*



(a) Rectilinear Mesh (b) Simplicial Mesh

Figure 2.1: Sample Meshes

The rectilinear mesh is generally the most natural mesh when data is sampled on a regular rectilinear grid (Assn. 1.2, p.7). However, there can be ambiguity in interpolation, notably when the Marching Cubes algorithm is used (Def. 3.1.2).

Definition 2.3 *A* simplicial mesh *is a mesh (Def. 2.1) in which each cell is a* simplex*, as in Fig. 2.1(b).*

In $\mathbb{R}^3$, a simplex is a tetrahedron (not necessarily regular).

As noted in Sec. 1.1, visualization techniques usually assume that the data is sampled from a continuous function defined over V. An *interpolation function* is then chosen to approximate the continuous function in V. This interpolation function is normally based on a mesh M that fills the volume V:

Definition 2.4 *An* interpolation function *is a function $f(x)$ defined over the volume V, with the following properties:*

Figure 2.2: A Level Set consisting of 3 Contours

1. Mesh-based: a mesh M (Def. 2.1, p.9) is chosen to fill the spatial volume V.

2. Local Interpolation: the function in any cell of the mesh is based solely on the values at the vertices of the cell.

The most commonly used interpolation function for simplicial meshes is the piecewise-linear function over the cells of the mesh: see Assn. 1.4, p.7. For rectilinear meshes, the most commonly used interpolation function is a tri-linear interpolation over the cells of the mesh.

Definition 2.5 *A* level set *of a function f at an* isovalue h *is the set* $L(h) = \{x \in Dom(f) : f(x) = h\}$.

Note that a level set may be empty, or may consist of one or more connected components, or *contours* (Def. 2.6), as shown in Fig. 2.2.

Definition 2.6 *A* contour *is a connected component of a level set (Def. 2.5).*

In the literature, *isosurface* is sometimes used to refer to a level set, and sometimes to a contour. In order to avoid confusion, I avoid the use of "isosurface", and use only "level set" and "contour".

2.1 Parameters for analysis

In analyzing algorithms for generating level sets, I use a number of parameters:

Definition 2.7 *n is the number of vertices in the mesh(Def. 2.1) being used.*

Definition 2.8 *N is the number of cells in the mesh.*

All algorithms considered use either simplices or voxels as cells in $\mathbb{R}^3$. Both of these cell shapes have a fixed number of faces. Thus, the number of faces is $O(N)$. In fact, since each face can belong to only 2 cells, the number of faces is $\Theta(N)$.

Since an edge can belong to an arbitrary number of simplices, but each simplex has a fixed number of edges, it follows that the number of edges is $O(N)$. In a rectilinear mesh, which must have a fixed maximum number of cells containing an edge, the number of edges is $\Theta(N)$.

Similarly, cells have a fixed number of vertices, but vertices may belong to an arbitrary number of cells in a simplicial mesh, so $n = O(N)$. For a rectilinear mesh, or for a regular simplicial mesh, such as those discussed in Sec. 9.1, p.91, $n = \Theta(N)$.

Definition 2.9 *k is the size of the output.*

For maximum generality, k refers to the number of cells intersected by a given level set. In some situations, it is more convenient to think of k as the number of triangles generated for rendering. Since all algorithms considered in $\mathbb{R}^3$ generate at most 4 triangles per cell (see Fig. 3.1, p.15 and Fig. 3.5, p.22), these two measures are within a constant factor of each other.

Although $k = \Omega(N)$ in extreme cases, it is claimed that, for typical data in $\mathbb{R}^3$, $k = O(N^{2/3})$ [16, 15].

Definition 2.10 *t is the number of local extrema in the mesh M.*

A local extremum in the mesh M is a vertex x_i, all of whose neighbours either have smaller values than h_i (a local minimum - Def. 4.9, p.33), or larger values (a local maximum - Def. 4.9, p.33).

I will show in Sec. 4.5, p.35 that the size of the contour tree is $\Theta(t)$: thus, t can be used as a rough measure of the complexity of the dataset. In extreme cases, $t = \Omega(n)$: a turbulent fluid dataset from fluid dynamics might be one such case [17]. For X-ray crystallographic data, however, I expect that $t \ll n$.

Chapter 3

Prior Work

3.1 Marching Cubes

The simplest algorithm for generating level sets is the "Marching Cubes" algorithm of Lorenson and Cline [18]. This algorithm "marches" through all the cubes (i.e. voxels) in a rectilinear mesh, computing the intersection with the level set for each such voxel. For any voxel, each vertex is classified as "above" or "below" the level set, producing an 8-bit index into a lookup table. This lookup table, loosely based on a tri-linear interpolation function, defines how to compute the intersection (see Fig. 3.1).

Marching Cubes is simple and relatively easy to code. However, it has several disadvantages: running time, ambiguity, local coherence, and connectivity.

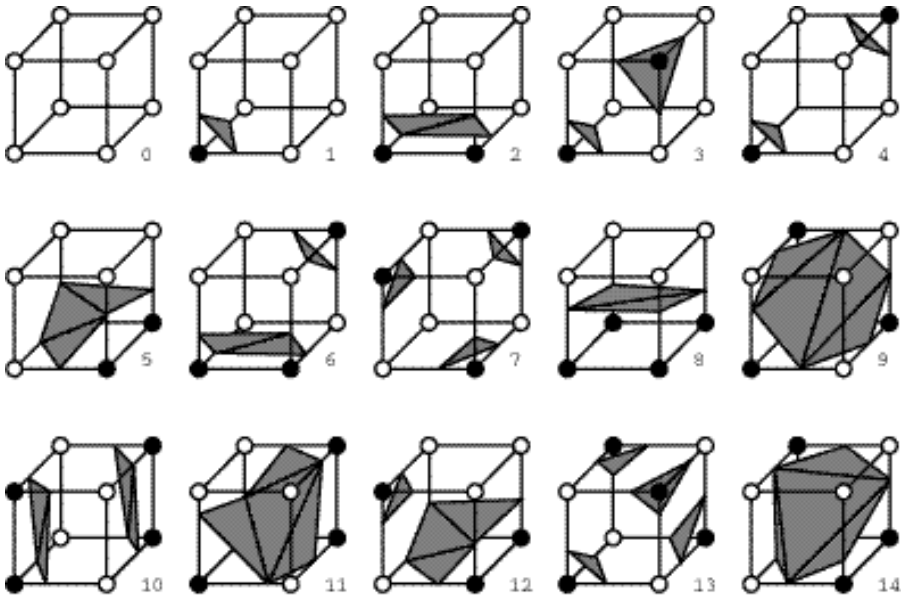


Figure 3.1: Marching Cube cases (after symmetry)

3.1.1 Analysis

Since Marching Cubes steps through all cells, it is trivial to analyze: the running time is $\Theta(N)$, and no preprocessing or additional space is required. Subsequent work has focussed on output-sensitive algorithms, which take advantage of the fact that $k = O(N^{2/3})$ (Def. 2.9) in typical cases. These algorithms normally require $O(N \log N)$ preprocessing time, although no lower bound has been proven.

Thus, Marching Cubes is preferred for generating single static level sets: alternate algorithms are preferred only when the preprocessing step can be amortized over multiple level sets, or when they provide additional information about the dataset.

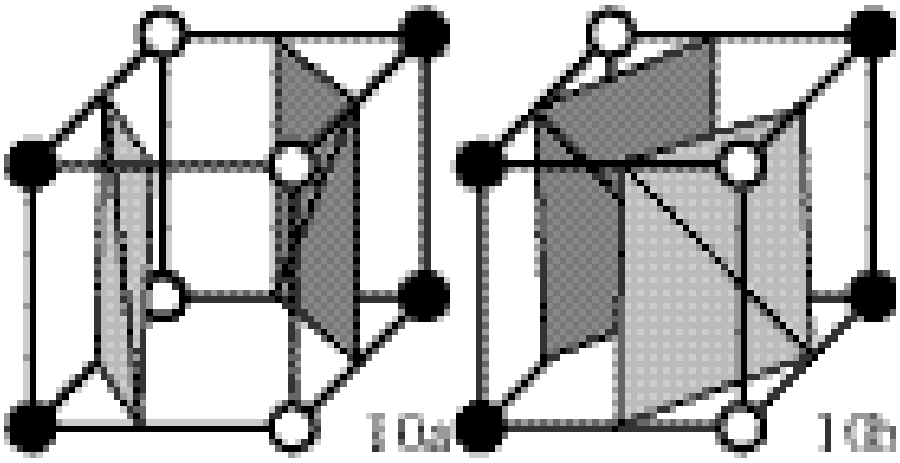


Figure 3.2: Ambiguous Marching Cube cases

3.1.2 Ambiguity

For trilinear interpolation, it is possible to construct voxels with identical Marching Cube cases, but topologically different surfaces (Fig. 3.2). Marching Cubes resolves this by assuming all such cases to be of one type, leading to topologically incorrect level sets and visual artefacts, although solutions for these problems were later identified by Nielson & Hamann [19].

This disadvantage does not exist with linear interpolation in simplices, for which no ambiguous cases exist (see Sec. 3.5.1 and Fig. 3.5).

3.1.3 Local Coherence

As Howie & Blake [14] point out, most graphics hardware can process strips of triangles more efficiently than individual triangles, since 2/3 of the vertices in a triangle are implicitly carried forward from the previous triangle. This can reduce the geometric calculations in the graphics hardware by a factor

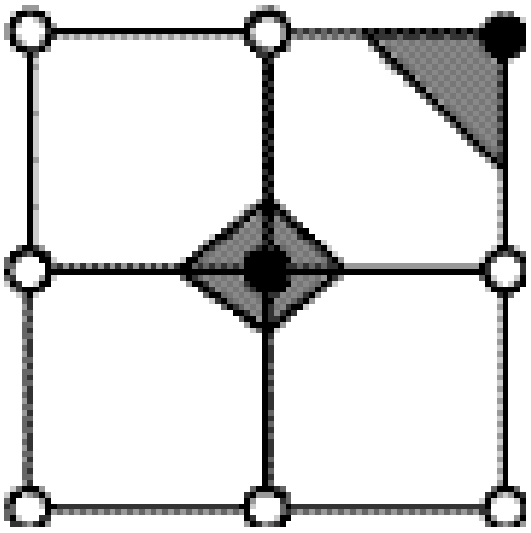


Figure 3.3: Two Contours in the Same Cell

of 3. Since there is frequently a bottleneck in transferring triangles to the graphics hardware, this consideration is of major importance, and algorithms that permit the generation of triangle strips are preferred. To achieve this, as many adjacent cells as possible should be processed sequentially.

Marching Cubes follows the grid axes: two voxels which are adjacent in the direction of travel will be processed sequentially. Thus, if the level set intersects the face between the two voxels, some vertices may be reused. However, for surfaces that do not follow the grid, this is not possible. Runs of cells from which triangle strips can be generated will tend to be short.

3.1.4 Connectivity

Marching Cubes cannot distinguish between separate contours in the level set, as no topological information is generated. In some cases, Marching Cubes

will actually render facets of two or more contours in the same voxel at the same time (as in Fig. 3.3). Where it may be desirable to distinguish between contours, as in X-ray crystallography, Marching Cubes is therefore at a disadvantage.

3.2 Octrees

Wilhelms & van Gelder [27] proposed using octrees to accelerate Marching Cubes. Under this scheme, each node in an octree is labelled with the minimum and maximum values of all voxels contained in the node's subtree. Level sets are then constructed by commencing at the root of the octree, and propagating downwards through all children spanning the desired isovalue.

Construction of the octree takes $O(N \log N)$time, and worst-case time to construct a level set is $O(k + \log n/k)$ [17].

Most disadvantages of Marching Cubes are retained (Sec. 3.1.2, 3.1.3, 3.1.4). In addition, Shen & Johnson [22] note that octree-based methods are vulnerable to noise in the data.

A variation on this has also been used by Bajaj and Pascucci [1] for progressive rendering of isosurfaces.

3.3 Span Space

Livnat, Shen & Johnson [17] consolidated much of the work on level sets, and proposed a new method based on Bentley's kd-trees [4]. They represent each

cell as the interval defined by the minimum and maximum values of the cell. The cell then intersects any level set whose isovalue falls into the interval. This reduces the problem to that of finding all intervals containing a desired isovalue. The intervals are recorded as (min, max) pairs, and the resulting two dimensional space (the *span space*) is searched using a kd-tree [17].

Construction of the kd-tree takes $O(N \log N)$ time and $O(N)$ space. Construction of a level set based on the kd-tree takes $O(\sqrt{n} + k)$ time, by retrieving all cells intersecting the desired interval.

This method has advantages other than just speed: it works for rectilinear and simplicial meshes, and on both regular and irregular grids. Local coherence (Sec. 3.1.3) is not achieved, however, since the kd-tree generates a list in which the cells are not necessarily related to each other. In this respect, span space may in fact be worse than Marching Cubes, since there is no guarantee that adjacent cells are ever processed sequentially. Similarly, the ability to identify different contours in a level set (Sec. 1.3, 3.1.4) is also lost.

3.4 Interval Trees

Cignoni et al. [5] follow Livnat, Shen & Johnson (Sec. 3.3) in treating each cell as an interval. These intervals are then stored in Edelsbrunner's *interval tree* [10], instead of the kd-tree. The interval tree is then used to search for cells intersecting the level set: otherwise the technique is identical to the Span Space approach.

Construction of the interval tree takes $O(N \log N)$ time and $O(N \log(N))$

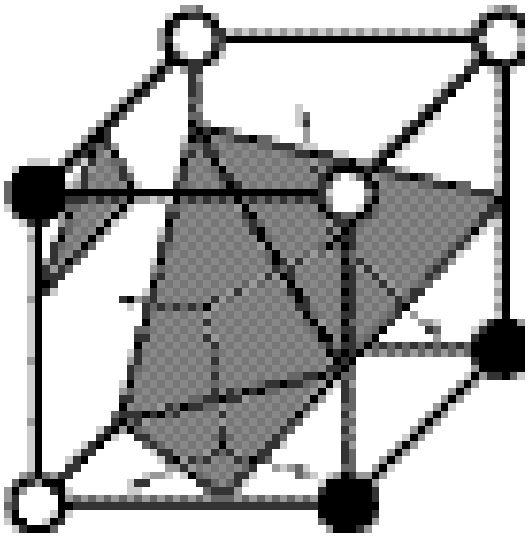


Figure 3.4: Contour Following

space: construction of a level set takes $O(k + \log n)$ time.

All other advantages and disadvantages of the span space approach are preserved.

3.5 Contour Following

All of the algorithms referred to so far lack local coherence (Sec. 3.1.3) and connectivity (Sec. 3.1.4). As a result, these methods generate a list of unrelated cells intersecting the level set. A class of algorithms exists which preserve both coherence and connectivity: I refer to these algorithms as *contour-following*[1], since they follow a contour from cell to cell.

In simple terms, the contour-following algorithm generates contours as follows:

[1]Other names include *continuation* [20], and *mesh propagation* [14]

1. Choose a cell that intersects with the level set.

2. Construct the surface of intersection of the cell and the level set.

3. Note that for each face of the cell that intersects with this surface, the adjacent cell must also intersect with the surface (see Fig. 3.4).

4. "Follow" the surface into each adjacent cell, by repeating steps 2 - 4 for that cell.

The surface constructed remains topologically connected at all times, since a cell is only visited if the surface under construction reaches into it from an adjacent cell. Thus, the algorithm constructs, not a level set, but a single component of the level set: a contour (Sec. 2.6). Hence the term "contour-following."

This was first implemented by Wyvill, McPheeters & Wyvill [28], using an explicit queue to visit the cells in a regular mesh, and was extended from regular to irregular meshes by Howie & Blake [14]. Both papers omit details on how to select an initial cell: in both cases, it appears that the initial cell was selected manually, or on the basis of domain-specific knowledge of the data set.

3.5.1 Seeds and Seed Sets

The contour-following algorithm stated above traces a single contour (i.e. connected component) of a level set. If the level set has more than one contour,

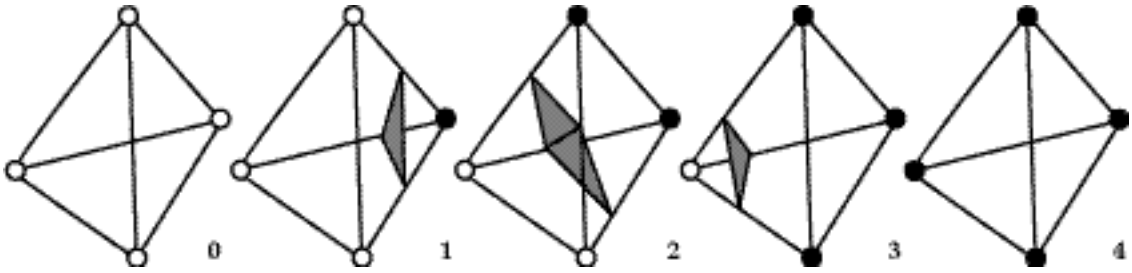


Figure 3.5: Possible Intersections of a Simplex and a Level Set

multiple starting points are needed to trace the entire level set. Such starting points are called *seeds*, and can be edges, faces, or cells. A set of seeds sufficient to generate an entire level set is called a *seed set*.

In a simplicial mesh, there are only 4 possible intersections of the cell and a level set not passing through a vertex: see Fig. 3.5. In each case, no more than one contour can intersect the simplex. Thus, each cell can be a seed for at most one contour at a given isovalue, and a seed set for a given level set must contain at least as many seeds as the level set contains contours.

If, the mesh is rectilinear, more than one contour can intersect a given voxel (e.g. Fig. 3.3). In this case, a cell may be a seed for more than one contour, and the seed set need not have as many seeds as there are contours in the level set. This complicates following the contours, but does not preclude it.

3.5.2 Analysis

Using contour-following to generate level sets requires $\Theta(k)$ time to generate all contours from seeds, plus the time to find the seeds. Since contour-following

itself requires no preprocessing, the preprocessing cost is simply the cost of generating any structures necessary to obtain seed sets for any desired isovalue.

Since contour-following expressly moves from one cell to the next across a shared boundary, it will usually produce connected triangles, thus preserving local coherence (see Sec. 3.1.3). In addition, since contour-following generates each contour separately, it can be used to distinguish contours in the level set, unlike Marching Cubes (see Sec. 3.1.4). Ambiguity (Sec. 3.1.2) is still a problem in non-simplicial meshes.

3.5.3 Extrema Graphs

Itoh & Koyamada [16, 15] use *extrema graphs* to generate seeds for contour-following. They preprocess the data to extract all local extrema, then connect pairs of extrema with arcs to form the extrema graph. Each edge of the extrema graph has an associated list of cells, in sorted order from a maximum to a minimum: one of these is chosen as a seed for any given isovalue.

No exact analysis is provided, but the authors admit that it is not always efficient, and is not guaranteed to succeed. Livnat, Shen & Johnson [17] observe that the worst case behaviour for generating a level set is $\Omega(n)$, as Itoh & Koyamada test all edges in the extrema graph for intersection with the isovalue.

3.5.4 Thinning

Bajaj et al. [2] have described a technique in which a set of cells is chosen as a seed set, then stored in a segment tree: generation of contours is then done by contour-following. To choose the seed set, the entire set of cells is initially chosen, then redundant cells are removed by a heuristic, thus reducing, or thinning the seed set, until an almost-optimal set is achieved.

In the same paper, Bajaj et al. also describe a greedy "climbing" algorithm which approximates the contour tree, and a sweep algorithm for constructing seed sets offline.

Chapter 4

Contour Trees

The algorithms discussed in the previous chapter were designed for 3-Ddatasets. Similar algorithms exist for 2-D datasets. For example, van Kreveld used interval trees (Sec. 3.4, p.19) to generate *contour lines* in 2-D [25]. In 1993, de Berg and van Kreveld [8] applied a structure called the *contour tree* in 2-D, and demonstrated that it could be constructed in $O(N \log N)$ time. In 1997, van Kreveld et al. [26] improved the algorithm for constructing the contour tree, and observed that it could be applied to higher dimensions, albeit with a construction time of $O(N^2)$. Their algorithm is discussed in detail in Sec. 4.8, as is a later result by Tarasov & Vyalyi, which extends the $O(N \log N)$ time bound to three dimensions [23].

4.1 Morse Theory

Contour trees are based, in part, on work in the field of Morse theory [3, 12]. Morse theory studies the changes in topology of level sets of f as the parameter x changes. Points at which the topology of the level sets change are called *critical points*. Morse theory requires that the critical points be isolated – i.e. that they occur at distinct points and values. A function that satisfies this condition is called a *Morse function*. All points other than critical points are called *regular points* and do not affect the number or genus of the components of the level sets.

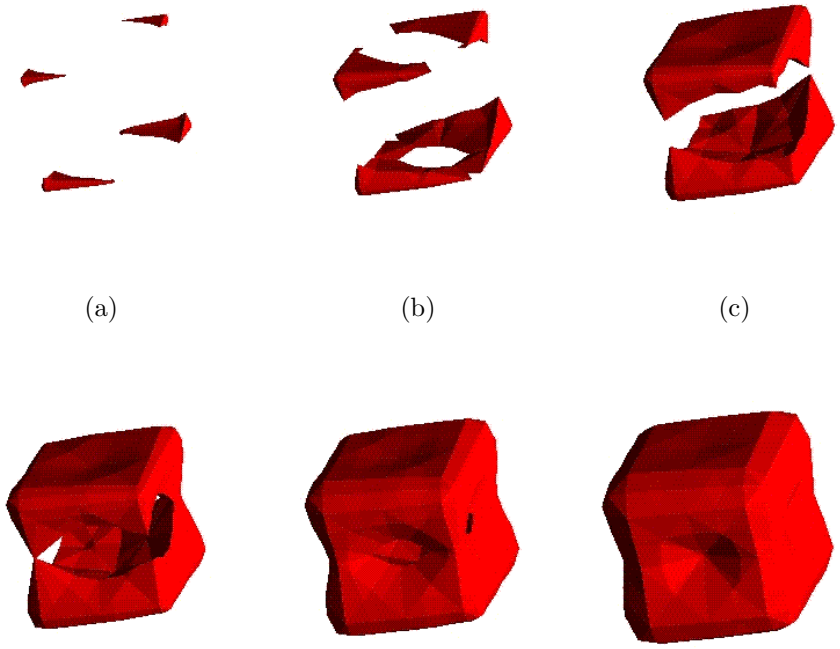
4.1.1 A Guarantee that Critical Points are Vertices

Morse theory provides some useful theoretical results. In the case of simplicial meshes, the definition of f – as a linear interpolant over a simplicial mesh with unique data values at vertices (Sec. 1.4) – ensures that f is a Morse function, and that the critical points occur at vertices of the mesh [3].

A direct proof of this result is given in the following sections. This proof does not rely on Morse theory, but is based solely on properties of the interpolation function.

4.2 Description of Contour Tree

If we think of the parameter x as time and watch the evolution of the level sets of f over time, then we see contours appear, split, change genus, join,



(a) (b) (c)

(d) (e) (f)

Figure 4.1: Development of Level Sets in 3-D

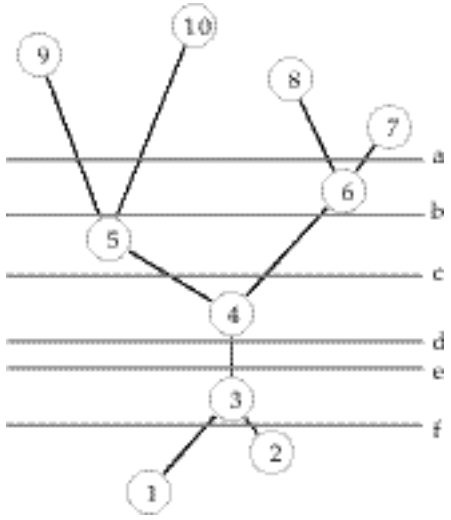


Figure 4.2: Contour Tree corresponding to Fig. 4.1

and disappear. In the example shown in Fig. 4.1, the level set evolves from four sticks, to two rings, to two cushions, to a hollow ball, to a solid, as the parameter decreases.

The *contour tree* is a graph that tracks contours of the level set as they split, join, appear, and disappear. Fig. 4.2 shows the contour tree for the dataset illustrated in Fig. 4.1. Starting at the global maximum, four small contours appear in sequence (10, 9, 8, 7): these correspond to the four leaves at the top of the contour tree. The surfaces join (6, 5) in pairs, forming larger contours, which quickly become rings. These rings then flatten out into cushions, which join (4) to form a single contour. This contour gradually wraps around a hollow core, and pinches off at (3), splitting into two contours: one faces inwards, the other outwards. The inward contour contracts until it disappears at (2): the outward contour expands until it reaches the global minimum (1).

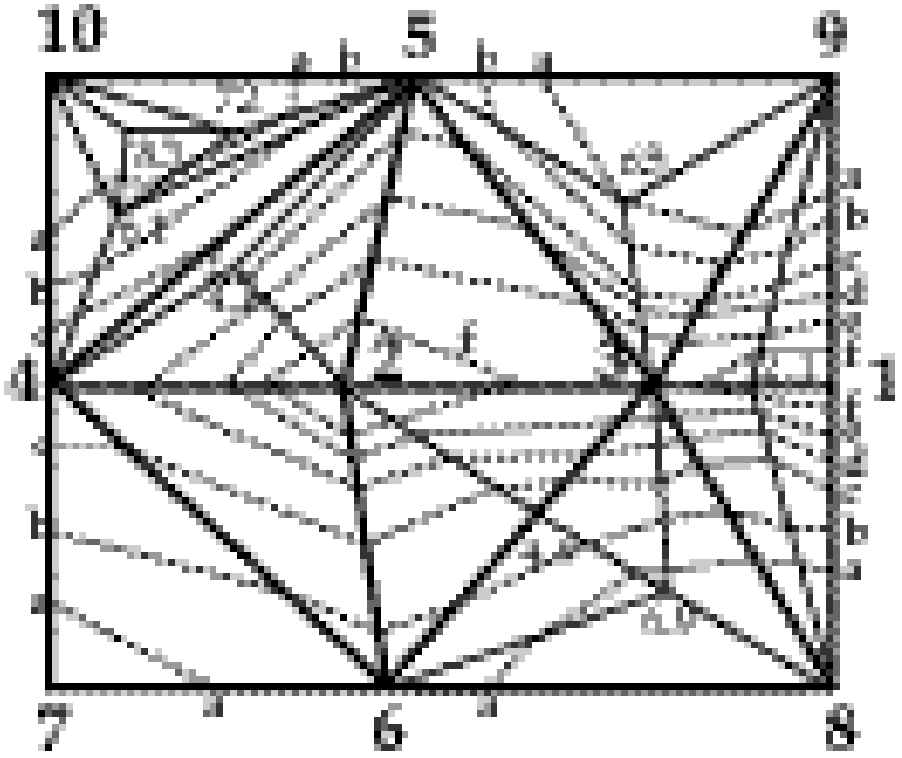


Figure 4.3: Development of Level Sets in 2-D

Note that changes in genus from disk to torus to disk are not reflected in the contour tree — even though the contours change topology, each can still be traced from a single seed point. All changes to the contours occur at critical points, but not all critical points cause changes to the contour tree. Formal definitions to support this intuitive description are found in the following sections.

Each "contour" referred to as appearing, joining, splitting and disappearing is actually a *class of contours* for different isovalues, and I define contour trees in terms of equivalence classes of contours.

The contour tree is independent of the dimension of the space - for

example, Fig. 4.3 shows a dataset in 2-D which has Fig. 4.2 as its contour tree. In some of the examples that follow, I shall use this contour tree instead of Fig. 4.1, where the desired property is clearer in 2-D than in 3-D.

4.3 Contour Properties

In order to define contour trees, I start by showing that contours are essentially unchanged between isovalues of vertices of the mesh. From Assn. 1.5, the vertices $x_1, x_2, \ldots, x_n$ must have unique values: it is convenient for contour tree computation if they are in sorted order. To simplify some definitions, add $h_0 = -\infty$ and $h_{n+1} = \infty$:

Assumption 4.1 $h_0 < h_1 < h_2 < \ldots < h_n < h_{n+1}$

Definition 4.2 *The* contour containing a point p, *denoted* γ_p, *is the contour* γ *to which* p *belongs.*

Definition 4.3 *For a contour* κ *at isovalue* h, *define* the set of simplices intersected by κ, *to be* $Simp(\kappa) = \{\sigma$ *is a simplex in the mesh* $: \sigma \cap \kappa \neq \emptyset\}$ *(see Fig. 4.4).*

The goal is to construct a contour tree over the entire spatial volume V, based on the values h_i at the vertices of the mesh: the first step towards this goal is to show that changes in the connectivity of the level set can only happen at vertices of the mesh. I start by showing that changes in the connectivity cannot happen between successive values h_i, h_{i+1}:

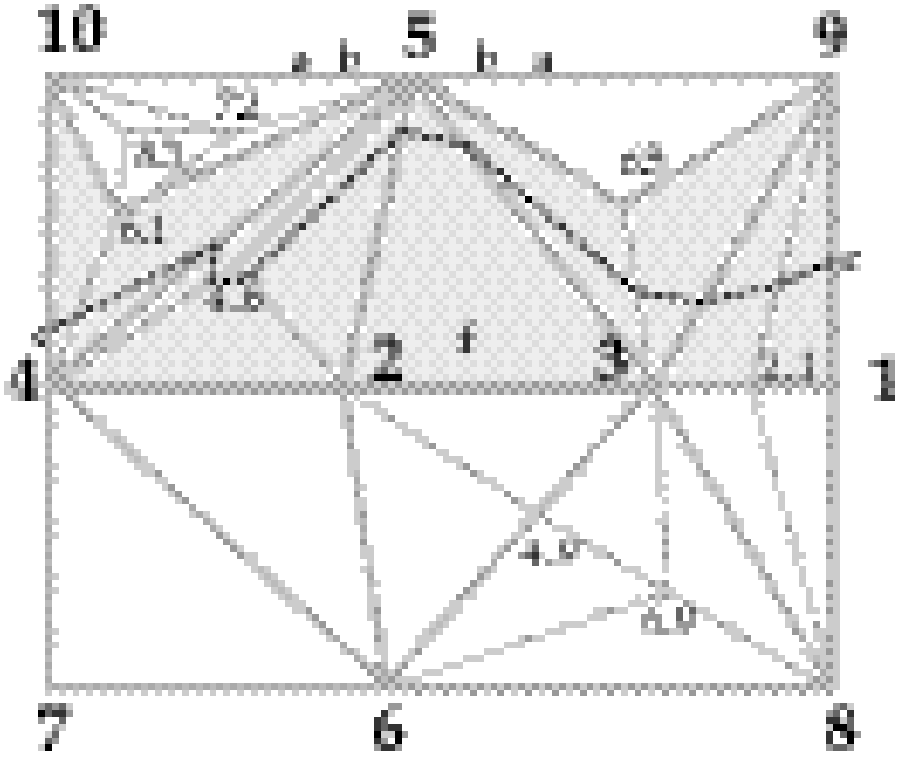


Figure 4.4: Simplices Intersecting a Contour

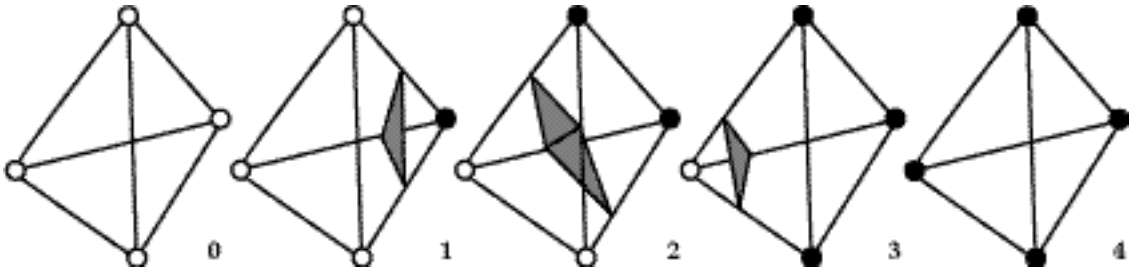


Figure 4.5: Possible Contours in a Simplex (from Fig. 3.5, p.22)

Lemma 4.1 *Let κ, λ be contours at distinct isovalues $h, h' \in (h_{i-1}, h_i)$, and $Simp(\kappa) \cap Simp(\lambda) \neq \emptyset$. Then $Simp(\kappa) = Simp(\lambda)$.*

Proof: Since κ is connected, all the simplices in $Simp(\kappa)$ must be connected. Assume that $Simp(\kappa) \neq Simp(\lambda)$. Then there must be at least one simplex σ_1 in $Simp(\kappa) \cap Simp(\lambda)$ from which κ continues through a face of σ_1 into some other simplex σ_2 which is in $Simp(\kappa)$ but not $Simp(\lambda)$ (or vice versa). Without loss of generality, assume the former.

There must then be a edge τ, common to σ_1 and σ_2, that intersects κ. Let x_j and x_k be the two vertices which τ connects, with $j < k$. Then $h_j < h < h_k$, by the assumption that the interpolation function is piecewise linear (Assn. 1.4, p.7). Also, $h_{i-1} < h < h_i$, and the vertices are in sorted order, so $h_j \leq h_{i-1} < h < h_i \leq h_k$. Then $h_j < h' < h_k$, since $h_{i-1} < h' < h_i$. By the Mean Value Theorem, since f is continuous over each simplex, there exists some point x on the edge τ such that $f(x) = h'$. Since only one contour can intersect a given simplex (Sec. 3.5.1, p.22), this means that $\lambda \cap \tau \neq \emptyset$. But τ is contained in σ_2, so $\lambda \cap \sigma_2 \neq \emptyset$, i.e. $\sigma_2 \in Simp(\lambda)$, which contradicts our assumption that σ_2 was in $Simp(\kappa)$ but not $Simp(\lambda)$.□

Note that this result will not hold in a rectilinear mesh, where two topologically separate contours can intersect the same cell (Sec. 3.1.2, p.16).

A consequence of this lemma is that we can continuously transform a contour at height h to a new contour at some nearby height h':

Corollary 4.2 *Let κ be a contour at isovalue $h \in (h_{i-1}, h_i)$, and let $h' \in (h_{i-1}, h_i)$. Then there is exactly one contour λ at isovalue h' such that $Simp(\kappa) = Simp(\lambda)$.*□

Proof: Let σ be any simplex in $Simp(\kappa)$. Then there is some x in σ such that $f(x) = h$. By the same argument as used in Theorem 4.1, there must also be some x' in σ such that $f(x) = h'$. Call the contour to which x belongs λ. Then, by Theorem 4.1, $Simp(\kappa) = Simp(\lambda)$. Since only one contour for a given isovalue can intersect σ, λ must be unique.$\Box$

Clearly, little of interest occurs between isovalues of the vertices: see Fig. 4.5. We classify each vertex x_i by the behaviour of the contours that are just above and just below x_i, for some small $\epsilon > 0$ such that $h_{i-1} < h_i - \epsilon < h_i < h_i + \epsilon < h_{i+1}$.

Definition 4.4 *The* star of a vertex x_i, *denoted* $Star(x_i)$, *is the set of simplices that have* x_i *as one of their vertices.*

Definition 4.5 *The* set of contours just above x_i*:*

$$C_i^+ = \{\gamma \in L(h_i + \epsilon) : \gamma \ \cap \ Star(x_i) \neq \emptyset\}.$$

Definition 4.6 *The* set of contours just below x_i*:*

$$C_i^- = \{\gamma \in L(h_i - \epsilon) : \gamma \ \cap \ Star(x_i) \neq \emptyset\}.$$

Now classify the vertices of the mesh as follows:

Definition 4.7 *An* ordinary point *is a vertex* x_i *for which* $\|C_i^+\| = 1$ *and* $\|C_i^-\| = 1$.

Definition 4.8 *A* local maximum *is a vertex* x_i *for which* $\|C_i^+\| = 0$.

Definition 4.9 *A* local minimum *is a vertex* x_i *for which* $\|C_i^-\| = 0$.

Note that this definition of local extrema is different from the usual definition of local extrema in a graph: vertices larger [smaller] than all their neighbours. However, the two definitions are equivalent: this definition was chosen to provide a consistent test for extrema and other critical points.

Definition 4.10 *A* join *is a vertex x_i for which $\|C_i^+\| > 1$.*

Definition 4.11 *A* split *is a vertex x_i for which $\|C_i^-\| > 1$.*

Definition 4.12 *A* critical point *is a vertex x_i which is a local maximum, local minimum, join, or split.*

Note that a vertex may be both a join and a split, a join and a local minimum, or a split and a local maximum. These cases normally occur only on the boundary of the volume, and not in the interior. All other possibilities are mutually exclusive.

Note that this definition of *critical point* is not quite the same as the definition of critical point in Morse theory: I treat a vertex where only the topological genus changes as an ordinary point.

4.4 Contour Equivalence

In this section, I define an equivalence relation that captures the intuitive description of the contour tree. The contour tree will then be defined in terms of the equivalence classes of this relation.

Definition 4.13 *Let γ, γ' be contours at h, h', respectively, with $h < h'$. Then γ, γ' are* contour equivalent *($\gamma \equiv \gamma'$) if all of the following are true:*

1. *neither γ nor γ' passes through a critical point,*
2. *γ, γ' are in the same connected component Γ of $\{x : f(x) \geq h\}$, and there is no join $x_i \in \Gamma$ such that $h < h_i < h'$, and*
3. *γ, γ' are in the same connected component Δ of $\{x : f(x) \leq h'\}$, and there is no split $x_i \in \Delta$ such that $h < h_i < h'$.*

Definition 4.14 *The equivalence classes of this relation will be called* contour classes.

Definition 4.15 *The* contour class containing a point *p, denoted $[\gamma_p]$, is the contour class to which the contour γ_p belongs (recall that γ_p is the contour to which p belongs.*

4.5 Definition of the Contour Tree

The definition of contour classes encapsulates the intuitive "sweep" described in Sec. 4.2. Contours not passing through critical points will belong to contour classes that map 1-1 with open intervals (h_i, h_j), where x_i, x_j are critical points, and $i < j$. I will sometimes refer to a contour class as being *created* at j, h_j, or x_j, and being *destroyed* at i, h_i, or x_i, thus preserving the intuitive description of a sweep from high to low values.

Contours that pass through critical points will be the sole members of the contour classes to which they belong (i.e. finite contour classes).

This correspondence between critical points and finite contour classes, and between open intervals and infinite contour classes, leads to the definition of the contour tree for a simplicial mesh:

Definition 4.16 *The* contour tree *is a graph composed of:*

1. *vertices, or* supernodes*, which represent finite contour classes (i.e. critical points):*
   (a) *Leaf vertices, corresponding to:*
      (i). *local maxima, at which an infinite contour class is created*
      (ii). *local minima, at which an infinite contour class is destroyed*
   (b) *Interior vertices, at which*
      (i). *at least one infinite contour class is created, and*
      (ii). *at least one infinite contour class is destroyed*
2. *edges, or* superarcs*, which represent infinite contour classes, and connect:*
   (a) *the supernode corresponding to the critical point at which the contour class is created, and*
   (b) *the supernode corresponding to the critical point at which the contour class is destroyed.*

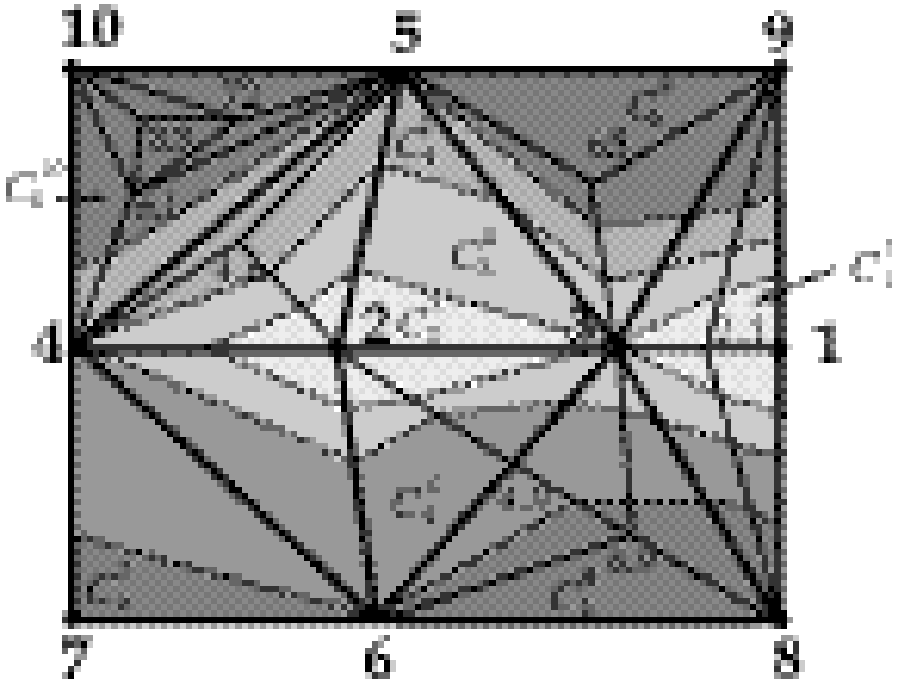


Figure 4.6: Critical Points and Boundaries Between Regions

Label each superarc $\mathcal{C}_i^j$, where x_j is the vertex at which the superarc is created, and x_i is the vertex at which it is destroyed. Fig. 4.6 shows the correspondence between the contour tree in Fig. 4.2 and regions of Fig. 4.3: the correspondence to Fig. 4.1 is similar, but harder to show on paper.

4.6 Properties of the Contour Tree

The first useful property of the contour tree is that it is in fact a tree. The proof of this is taken from de Berg and van Kreveld [8], with minor modifications, and is included for completeness:

Lemma 4.3 *The contour tree C for a connected spatial volume is a tree.*

Proof: Proof is by showing that the contour tree is acyclic, and is connected if the volume V is connected.

Connectedness: Let x and y be points in V that are connected by some path P. Each point p on P belongs to some contour γ_p, which in turn belongs to some contour class $[\gamma_p]$, which may be either finite or infinite.

Let $P' = \{\gamma_p : p \in P\}$, i.e. P' is the set of contours containing points on P. Then, since each γ_p contains the corresponding p, P' contains P. Therefore P' connects x and y. Note that if p and q are points on P with the same isovalue $f(p) = f(q)$, then γ_p and γ_q may in fact be the same contour. If this happens, I reduce P' by omitting all contours for points between p and q on P. Thus, I assume that P', viewed as a set of contours, contains no repeats.

Let $P'' = \{[\gamma_p] : p \in P\}$, i.e. P'' is the set of contour classes containing points on P. Since each $[\gamma_p]$ contains the corresponding γ_p, and each γ_p contains the corresponding p, the set P'' must also contain P. Therefore P'' connects x and y. As with P', I assume that there are no duplicates: once P exits a contour class, it never re-enters that contour class.

Consider the set Q consisting of all supernodes and superarcs that correspond to contour classes in P''. Clearly, Q contains both $[\gamma_x]$ and $[\gamma_y]$. I claim that Q connects $[\gamma_x]$ and $[\gamma_y]$ in the contour tree.

Let $\mathcal{C}_i^j$ be a superarc in the contour tree, corresponding to some infinite contour class in P'', and let x_i and x_j be the vertices at which $\mathcal{C}_i^j$ is created and destroyed, respectively.

Note that the connectivity of the level set can only change at critical points (Sec. 4.1.1), and an infinite contour class contains no critical points. This implies that the infinite contour class $\mathcal{C}_i^j$ consists of exactly one contour for each isovalue in the open interval (h_i, h_j). Also note that each of these contours is a connected component of the level set for that isovalue. Therefore, any path from a point in $\mathcal{C}_i^j$ to a point not in $\mathcal{C}_i^j$ must include at least one point x on either γ_{x_i} or γ_{x_j}. If we consider what this means in the contour tree, we see that if the path P leaves $\mathcal{C}_i^j$ in P'', then either x_i or x_j belongs to Q. Assume that it is x_i, and note that $\mathcal{C}_i^j$ is connected to x_i in the contour tree. Similarly, when the path P leaves γ_{x_i}, it must enter some new infinite contour class, $\mathcal{C}_i^k$ or $\mathcal{C}_k^i$, to which x_i is connected in the contour tree. Continuing in this way, we see that the sequence of contour classes along the path P corresponds

exactly to a set of connected supernodes and superarcs in the contour tree: i.e. Q is connected.□

It follows from this that, if the volume is connected, so is the contour tree.

Acyclicity: Suppose a cycle K exists in the contour tree. Without loss of generality, assume that K has no repeated supernodes, and pick the smallest-valued supernode x_i on the cycle: two distinct superarcs $\mathcal{C}_i^j$ and $\mathcal{C}_i^k$ are incident to x_i. Since $\mathcal{C}_i^j$ represents contours for all isovalues in the open interval (h_i, h_j), I choose a point x on contour γ at isovalue h such that $[\gamma] = \mathcal{C}_i^j$, and $h_i < h < h_{i+1}$. Similarly, I choose a point y on contour γ' at isovalue h' such that $[\gamma'] = \mathcal{C}_j^i$, and $h < h' < h_{i+1}$.

Note that neither γ nor γ' passes through a critical point. Thus, γ and γ' satisfy the first condition for contour-equivalence (Def. 4.13).

Let Q be the open-ended path obtained by removing x_i from K without deleting $\mathcal{C}_i^j$ and $\mathcal{C}_i^k$. By an argument similar to that used for connectivity above, this path Q must correspond to a connected set in V consisting solely of points with isovalues $> h_i$, and including x and y. I truncate this connected set by removing all points with isovalues $< h$. Since $h_i < h < h_{i+1}$, and changes to connectivity can only occur at vertices (Sec. 4.1.1), this does not disconnect the set.

Since x and y sit on contours γ and γ' respectively, it then follows that γ and γ' belong to the same connected component Γ of $\{x : f(x) \geq h\}$. And since $h_i < h < h' < h_{i+1}$, there is no join $x_j \in \Gamma$ such that $h < h_j < h'$,

satisfying the second condition of Def. 4.13.

Finally, let R be the open-ended path in K obtained by taking x_i, $\mathcal{C}_j^i$ and $\mathcal{C}_k^i$. A similar argument to that just used then shows that γ and γ' are in the same connected component Δ of $\{x : f(x) \leq h'$, and there is no split $x_j \in \Delta$ such that $h < h_j < h'$, satisfying the third condition of Def. 4.13.

Thus, $\gamma \equiv \gamma'$, and it follows that $\mathcal{C}_i^j = \mathcal{C}_i^k$, so by contradiction, no cycle K exists in the contour tree.□

Corollary 4.4 *The contour tree C for the volume V is of size $\Theta(t)$.*

Proof: From Def. 4.8 and Def. 4.9, it is clear that the leaves of the contour tree must be local extrema. By Def. 2.10, p.13, there are t local extrema. Since at least half of the nodes of a tree must be leaves, the size of the contour tree is at least $t + 1$ and at most $2t - 1$.□

4.7 Augmented Contour Tree

For some purposes, we may wish to know more information than that provided by the contour tree. In particular, we may wish to know which non-critical points in the mesh belong to which superarc. One method of doing this is to augment the superarcs with all of the corresponding vertices, as follows:

Definition 4.17 *The* augmented contour tree, *AC_M of the mesh M is the tree on the vertices of M such that x_i and x_j are adjacent in AC_M if*

1. *x_i and x_j are both supernodes of the contour tree, $\mathcal{C}_i^j$ is a superarc of the contour tree, and there is no vertex x_k such that $[\gamma_{x_k}] = \mathcal{C}_i^j$,*

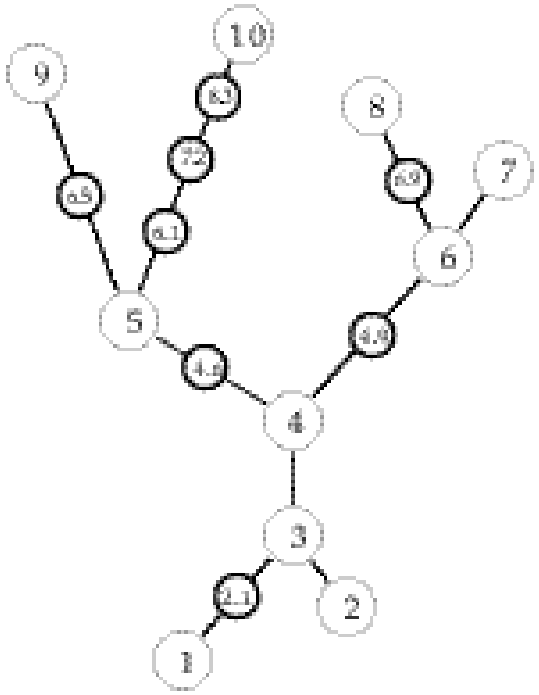


Figure 4.7: The Augmented Contour Tree corresponding to Fig. 4.3

2. *x_i is a supernode of the contour tree, x_j is not a supernode of the contour tree, $[\gamma_{x_j}] = \mathcal{C}_k^i$ and there is no vertex x_m such that $[\gamma_{x_m}] = \mathcal{C}_k^i$, and $h_j < h_m < h_i$,*

3. *x_i is a supernode of the contour tree, x_j is not a supernode of the contour tree, $[\gamma_{x_j}] = \mathcal{C}_i^k$ and there is no vertex x_m such that $[\gamma_{x_m}] = \mathcal{C}_i^k$, and $h_i < h_m < h_j$, or*

4. *x_i and x_j are not supernodes of the contour tree, $[\gamma_{x_i}] = [\gamma_{x_j}] = \mathcal{C}_m^k$, and there is no vertex x_p such that $[\gamma_{x_p}] = \mathcal{C}_m^k$, and h_m is strictly between x_i and x_j*

This definition strings the vertices belonging to a superarc out along that superarc in sorted order (see Fig. 4.7). Each superarc has become a path in the augmented contour tree. This path starts at the vertex where the superarc is created, passes through all of the vertices belonging to the superarc, and ends at the vertex where the superarc is deleted.

Lemma 4.5 *If x_i is an ordinary point in the mesh M, then x_i has exactly two neighbours x_j and x_k in AC_M, and $h_j < h_i < h_k$.*

Proof: Proof is by exhaustion of the cases in Def. 4.17. By Def. 4.7, $\|C_i^+\| = 1$ and $\|C_i^-\| = 1$. Thus x_i cannot be an endpoint of the superarc $\mathcal{C}_q^p$ of the contour tree to which x_i belongs. Consider x_p, the supernode at the top of $\mathcal{C}_q^p$, and suppose that there is no vertex x_k belonging to $\mathcal{C}_q^p$ such that $h_k > h_i$. Then by Def. 4.17(2), $x_i x_p$ is an edge of AC_M. Since x_i belongs to exactly one superarc, $\mathcal{C}_q^p$, there can be no other supernode that satisfies Def. 4.17(2). And since there is no x_k belonging to $\mathcal{C}_q^p$ that is higher than x_i, Def. 4.17(4) cannot connect x_i to any higher-valued vertex. This leaves two cases, both of which require a supernode for the lower-valued vertex. Thus the only higher-valued vertex connected to x_i is x_p.

If there is at least one vertex on $\mathcal{C}_q^p$ that is higher than x_i, let x_k be the smallest-valued such vertex: a similar argument to the foregoing shows that x_k is the only higher-valued vertex connected to x_i.

The same arguments then apply symmetrically to show that x_i has exactly one smaller-valued neighbour in AC_M.□

4.8 Contour Tree Algorithms

van Kreveld et al. [26] construct a contour tree on a simplicial mesh. Their construction assumes that saddle points are *simple* (i.e. involve no more than two contours splitting or joining). Multiple saddle points are split into multiple simple saddle points.

The algorithm proceeds by sweeping through the values of the parameter, maintaining the level set at all times as a set of contours, each contour being represented as a polygon. As the sweep passes through each vertex in the mesh, the level set is updated locally. If the vertex is a (simple) saddle point, either two contours join or two contours separate. In either case, a new supernode is generated, and the superarcs corresponding to each contour are set to terminate at the supernode. If two contours join at the saddle point, the level set is updated in time proportional to the smaller of the two contours. Similarly, a split is achieved in time proportional to the smaller of the two resulting contours. This allowed van Kreveld et al.to prove an upper bound of $O(N \log N)$ time in two dimensions. The same algorithm in higher dimensions resulted in a lower bound of $O(N^2)$ time, primarily due to the increased cost of maintaining the level set in dimensions above two.

Tarasov & Vyalyi [23] extend the result of van Kreveld et al. [26] to achieve a time of $O(N \log N)$ in three dimensions. Again, the mesh is assumed to be simplicial, and saddle points are assumed to be simple. The single sweep in [26] is replaced by three sweeps. Two sweeps are performed to identify local minima and maxima respectively: the third sweep then identifies and resolves

saddle points in a way similar to [26]. Boundary cases are handled by special cases within the algorithm.

Tarasov & Vyalyi's assumption that saddle points are simple is justified by a preprocessing step. This step involves a barycentric subdivision of each simplex into 24 smaller simplices, and may also involve further subdivisions. As a result, the size of the input data is magnified by a factor of 24 in all cases, and 360 in the worst case. When combined with the cost of simplicial subdivision (Sec. 9.1, p.91), this leads to a total magnification between 120 and 8640 for rectilinear data: this is prohibitive in practice.

I propose a new algorithm (Algorithm 7.2, p.84) for computing contour trees over simplicial meshes with:

1. Time requirements of $O(n \log n + N + t\alpha(t))$ for constructing contour trees, in any number of dimensions (Corollary 7.5, p.85),

2. Space requirements of $O(n)$ working storage (Theorem 7.4, p.84),

3. Simple treatment of boundary effects, and

4. Simple treatment of multi-saddle points.

Before giving details of the algorithm, some preliminary results are necessary. Ch. 5 deals with reconstructing trees from partial information. Ch. 6 then describes how to extract sufficient information from the data set to perform the reconstruction. Finally, Ch. 7 puts all the pieces together to produce an algorithm.

Chapter 5

Join Trees and Split Trees

This chapter presents some graph-theoretic results which form the basis of the contour-tree construction algorithm in Ch. 7.

Recall that Def. 4.13, p.35, defined contour equivalence in terms of components of the sets $\{x : f(x) \geq h\}$ and $\{x : f(x) \leq h\}$. Def. 4.16, p.36 then defined the contour tree in terms of classes of equivalent contours over the volume V. These contour classes reflect the underlying continuity of the volume V.

In a graph, such as the mesh M underlying the interpolation function, equivalence classes are rather less meaningful. However, connectivity still exists, and it is possible to define structures similar to the contour tree using only the sets $\{x : x \text{ is a vertex and } f(x) \geq h\}$ and $\{x : x \text{ is a vertex and } f(x) \leq h\}$. The information represented by these sets is collected separately in two structures called the *join tree* and the *split tree*.

This chapter contains two results: the reconstruction of an unknown

tree from its known join and split trees, and a demonstration that the join tree and split tree for the mesh M are identical to the join and split trees for the augmented contour tree AC_M.

The following chapter (Ch. 6) gives an efficient algorithm to compute the join and split trees of the mesh. Ch. 7 then unites the results in this chapter and Ch. 6 to obtain the algorithm to construct the contour tree.

5.1 Definitions of Join and Split Trees

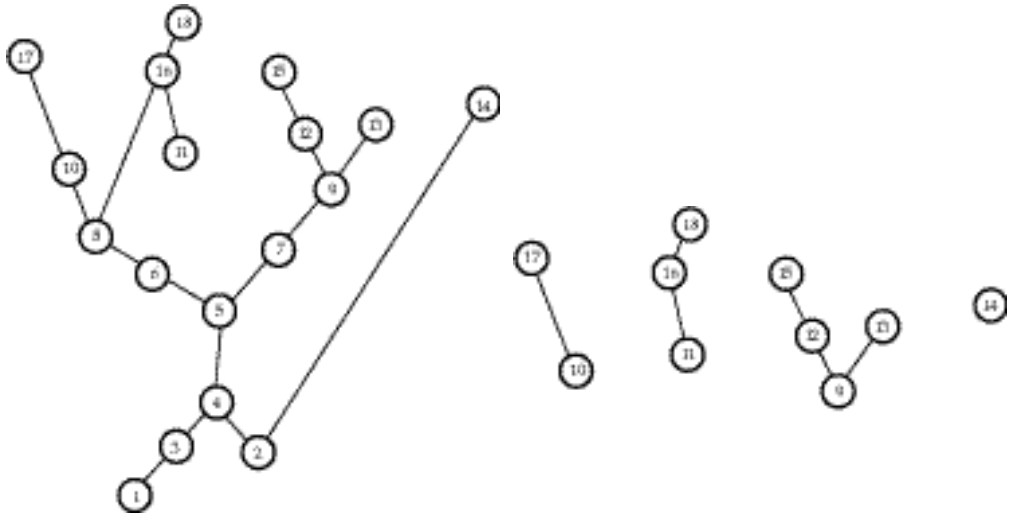
The join and split trees ignore the physical origin of the data: they are defined over an abstract class of graphs, called *height graphs*. For any given height graph, the *join tree* and *split tree* are then defined.

Definition 5.1 *A* height graph *is a graph G, each of whose n vertices x_i has a unique height h_i. As in Assn. 4.1, p.30, it is assumed that the vertices are in sorted order, i.e. $h_1 < h_2 < \ldots < h_n$.*

Definition 5.2 *A* C-tree *is a height graph C that is also a tree.*

Having defined height graphs, it is useful to consider what a height graph looks like "above" a given vertex (see Fig. 5.1), i.e. to consider the connectivity of the subgraph of all higher-valued vertices. This concept recurs so frequently that a formal definition is in order:

Definition 5.3 *The* subgraph of a height graph G above h_i, *denoted* $\Gamma_i^+(G)$, *is the subgraph of G induced by the vertices with height $> h_i$.*



(a) The C-Tree (b) The subgraph above x_8

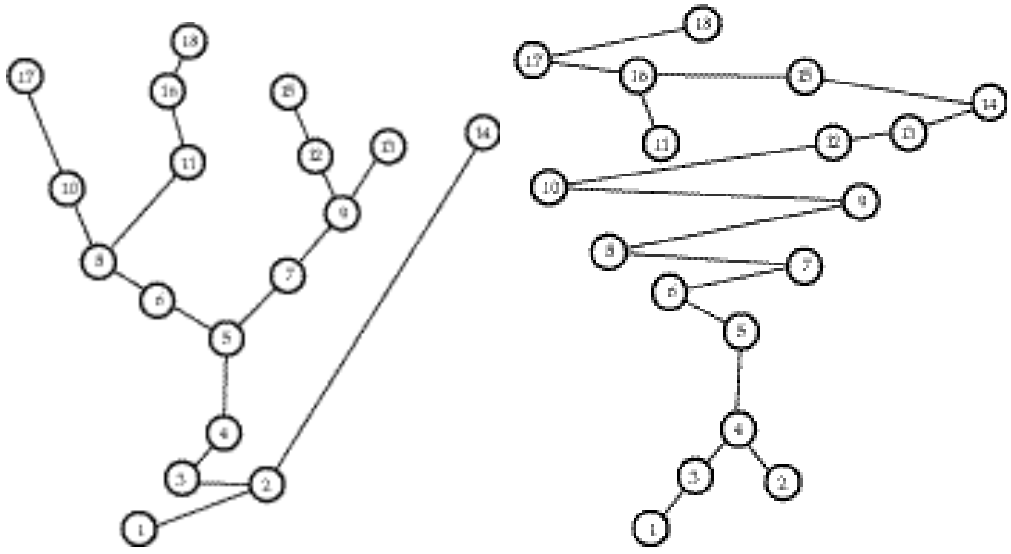
Figure 5.1: A sample C-Tree and subgraph above x_8

Definition 5.4 *The* subgraph of a height graph G below h_i, *denoted* $\Gamma_i^-(G)$, *is the subgraph of* G *induced by the vertices with height* $< h_i$.

I now define the join and split trees corresponding to a given height graph. Although the join and split trees are notionally on the same set of vertices as the height graph, the proofs become more intelligible if I adopt the convention that x_i is a vertex in a height graph (or C-tree), y_i is the corresponding vertex in the join tree, and z_i the vertex in the split tree (see Fig. 5.2 for the join and split tree corresponding to Fig. 5.1).

Definition 5.5 *The* join tree J_G *of a height graph* G *is a graph on the vertices* $y_1, \ldots, y_{\|G\|}$ *in which two vertices* y_i *and* y_j, *with* $h_i < h_j$, *are connected when:*

1. x_j *is the smallest-valued vertex of some connected component* γ *of* $\Gamma_i^+(G)$,



(a) The Join Tree (b) The Split Tree

Figure 5.2: Join and Split Trees corresponding to Fig. 5.1

2. x_i is adjacent in G to a vertex of γ, and

Definition 5.6 *The* split tree S_G *of a height graph G is a graph on the vertices $z_1, \ldots, z_{\|G\|}$ in which two vertices z_i and z_j, with $h_i > h_j$, are connected when:*

1. x_j is the largest-valued vertex of some connected component γ of $\Gamma_i^-(G)$,

2. x_i is adjacent in G to a vertex of γ, and

I now abandon height graphs temporarily, and work only with C-trees. In order to simplify proofs, I rely on the observation that the join and split trees are essentially the same, but for the direction of comparisons, so that:

Dual 5.1 *Any property that holds for join trees holds for split trees, with suitable modifications of up and down, higher and lower, $<$ and $>$, &c.*

All such properties will be noted as "Duals" to the relevant theorems and lemmas, without proof.

5.2 Vertex Degrees in Join and Split Trees

An important property of the join and split trees of a C-tree C is that the degree of a vertex in C can be deduced from its degree in J_C and S_C. This allows me to locate leaves of C without knowing C explicitly. Because the vertex heights are of critical importance, I differentiate between arcs leading "up" from a vertex, and those leading "down". Similarly, I distinguish between the "up-degree" and "down-degree."

Definition 5.7 *An* up [down] arc *is an arc from a vertex x_i to a higher-[lower-] valued vertex x_j.*

Definition 5.8 *The* up degree *of a vertex x_i, $\delta^+(x_i)$ is the number of up arcs at x_i.*

Definition 5.9 *The* down degree *of a vertex x_i, $\delta^-(x_i)$ is the number of down arcs at x_i.*

Once we have defined up and down arcs, it becomes meaningful to talk of local and global extrema:

Definition 5.10 *A* local maximum [minimum] *x_i in a height graph is a vertex with up [down] degree of* 0.

Definition 5.11 *The* global maximum [minimum] *of a height graph is the unique highest- [lowest-] valued vertex.*

Since leaves have only one arc, it is easy to see that a leaf must be an extremum. Either the arc is up, in which case the leaf is a local minimum, or the arc is down, in which case the leaf is a local maximum. But not all local extrema are leaves (for example, vertex 2 in Fig. 5.1): a vertex with up-degree 2 and down-degree 0 is obviously not a leaf. It turns out to be critical to differentiate between a leaf at the "top" of the C-tree, and a leaf at the "bottom", so I define:

Definition 5.12 *An* upper [lower] leaf *of a C-tree is a vertex with up- [down-] degree of 1, and down- [up-] degree of 0.*

Having defined my terms, I now prove some results about the degrees of vertices in the join and split trees. Recall that each vertex x_i in C has corresponding vertices y_i in J_C and z_i in S_C.

Theorem 5.2 *Given a C-tree C and its corresponding join tree J_C, $\delta^+(x_i) = \delta^+(y_i)$ for every vertex x_i in C and corresponding y_i in J_C.*

Proof: To prove this, I show that each up-arc x_ix_j in C corresponds 1-1 with an up-arc y_iy_k in J_C.

($\Rightarrow$): Let x_ix_j be an up-arc at x_i in C (see Fig. 5.3). Then x_j must belong to some connected component γ in $\Gamma_i^+(C)$. I claim that no other vertex x_l adjacent to x_i belongs to γ. Suppose there were another vertex x_l in γ

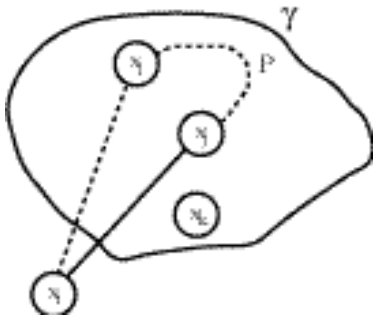


Figure 5.3: Components and Up-Arcs in the C-Tree

adjacent to x_i. Since γ is a connected component, a path P from x_j to x_l would have to exist in γ. Then $P + x_l x_i x_j$ would be a cycle in C. But C is a tree, and has no cycles, which contradicts the assumption that x_i is adjacent to more than one vertex in γ. Therefore, each up-arc at x_i corresponds to a component γ in $\Gamma_i^+(C)$.

Let x_k be the unique smallest-valued vertex in γ. By Def. 5.5, y_i is connected to y_k. Note that neither x_j nor x_k can belong to more than one connected component γ in $\Gamma_i^+(C)$. It follows that, for each up-arc at x_i, there exists at least one up-arc at y_i. □

($\Leftarrow$): Let $y_i y_k$ be an up-arc at y_i. By Def. 5.5, x_k belongs to some component γ in $\Gamma_i^+(C)$, and x_i is connected to some vertex x_j in γ. As with ($\Rightarrow$), this x_j must be unique, because C is a tree. Since $h_j > h_i$, we have shown that there is an up-arc at x_i in C corresponding to $y_i y_k$ in J_C.□

Dual 5.3 *Given a C-tree C and its corresponding split tree S_C, $\delta^-(x_i) = \delta^-(z_i)$ for every vertex x_i in C and corresponding z_i in S_C.*□

From this result, it is trivial to use properties of J_C and S_C to identify leaves of C:

Corollary 5.4 *If $\delta^+(y_i) = 0$ and $\delta^-(z_i) = 1$, then x_i is an upper leaf in C.*

From Theorem 5.2 and Corollary 5.3, $\delta^+(x_i) = 0$, and $\delta^-(x_i) = 1$. From Def. 5.12, it then follows that x_i is an upper leaf in C.□

Dual 5.5 *If $\delta^-(z_i) = 1$ and $\delta^+(y_i) = 0$, then x_i is a lower leaf in C.* □

If the up-degrees in the join tree and C-tree are identical, what can be said about the down-degree? Conveniently, the down-degree of a vertex in the join tree will always be 1, except at the global minimum, where it will be 0.

Lemma 5.6 *For each vertex y_j in the join tree J_C, $\delta^-(y_j) = 1$ unless $j = 1$, in which case $\delta^-(y_1) = 0$.*

Proof: Suppose that $\delta^-(y_j) > 1$. Then there are two down arcs at y_j, to y_i and y_k, with $h_i < h_k < h_j$. From Def. 5.5, x_j must be the smallest-valued vertex of some component γ of $\Gamma_k^+(C)$, and x_k is adjacent to some x_l in γ.

Again applying Def. 5.5, x_j belongs to some component ρ of $\Gamma_i^+(C)$. If x_m is a vertex of γ, it must be connected to x_j by a path whose vertices all have height $> h_j$ (else x_j would not be the smallest-valued vertex of γ). Since $h_i < h_j$, it follows that x_m belongs to ρ, and that $\gamma \subseteq \rho$. Because $x_l \in \gamma$, it is also true that $x_l \in \rho$, and since $x_k x_l$ is an arc between two vertices that are higher-valued than x_i, it follows that $x_k \in \rho$.

From Def. 5.5, x_j must be the smallest-valued vertex of ρ (otherwise $y_j y_i$ would not be an arc of J_C). But $h_k < h_j$, and x_k also belongs to ρ. By contradiction, it follows that x_j has down-degree ≤ 1.

To show that $\delta^-(x_j) = 1$, count the arcs in J_C and C. Since C is a tree on n vertices, it must have $n-1$ arcs. From Theorem 5.2, there are as many arcs in J_C as in C. Since no vertex may have down-degree > 1 in J_C, there must be $n-1$ vertices with down-degree 1, and one vertex with down-degree 0. Trivially, the global minimum, y_1, cannot have any down-arcs (since there are no vertices with smaller value), and $\delta^-(y_1) = 0$. Thus, if $j \neq 1$, $\delta^-(y_j) \neq 0$. □

Dual 5.7 *For each vertex z_j in the split tree S_C, $\delta^+(z_j) = 1$ unless $j = n$, in which case $\delta^+(z_n) = 0$.*□

Corollary 5.8 *The join tree J_C of a C-tree C is a tree, and also a C-tree.*

It is not difficult to see that J_C is connected: each vertex except y_1 must be connected to a lower-valued vertex. This vertex in turn connects to another vertex with smaller value yet: this can only stop when y_1 is reached: thus, each vertex is connected by a path leading downwards to y_1.

Suppose that there is a cycle K in J_C with unique highest-valued vertex y_i. Because K is a cycle, y_i must be adjacent to two other vertices of K, say y_j and y_k. Since y_i is the highest-valued vertex in K, the arcs $y_i y_j$ and $y_i y_k$ must be down-arcs. But this contradicts Lemma 5.6, which allows y_i to have at most 1 down arc.

Since J_C is both connected and acyclic, it must be a tree. And since each vertex has a unique height associated, J_C is also a C-tree.□

Dual 5.9 *The split tree S_C of a C-tree C is a tree, and also a C-tree.*□

5.3 Preliminaries to Reconstruction

The previous section showed that leaves of a contour tree C can be located using the up- and down- degrees in J_C and S_C. This can be extended to locate the incident arc to the leaf, then reduce J_C and S_C to smaller join and split trees. This permits reconstruction of C from J_C and S_C with a recursive algorithm.

The first step is to show that the incident arc to an upper [lower] leaf is present in the join [split] tree:

Lemma 5.10 *If x_i is an upper leaf, and y_iy_j is the incident arc to y_i in J_C, then x_ix_j is the incident arc to x_i in C.*

Proof: Let x_i belong to some component γ in $\Gamma_j^+(C)$. I claim that x_i is the only vertex in γ.

Suppose not. Then, since γ is a connected component, there is some other vertex x_k in γ to which x_i is connected. By Def. 5.5, x_i is the smallest-valued vertex in γ, so x_ix_k must be an up-arc at x_i. But, since x_i is an upper leaf, it has no up-arcs. It follows that x_i is the only vertex in γ.

Applying Def. 5.5, we see that, if y_iy_j is an arc of J_C, then x_j must be connected to some vertex in γ. But, since x_i is the only vertex in γ, it follows that x_j is connected to x_i.□

Dual 5.11 *If x_i is a lower leaf, and z_iz_j is the incident arc to z_i, then x_ix_j is the incident arc to x_i.*□

For the recursive construction of C, we remove a leaf x_i from C, and calculate the new join and split trees directly from the old ones. Although we aim to remove leaves, an upper leaf in C may not be a leaf in S_C. In this case, removing z_i from S_C may disconnect the split tree. Similarly, removing a lower leaf may disconnect J_C.

To avoid this, recall that the up-degree in S_C is always 1 (Dual 5.7). If the down-degree of x_i in S_C is more than 1, then the down-degree in C is also more than 1 (Dual 5.3), so x_i cannot be a leaf. Thus, x_i has exactly one up-arc in S_C, and either 0 or 1 down-arcs. Similarly, if x_i is a lower leaf, it has exactly one down-arc in J_C, and either 0 or 1 down-arcs.

In removing x_i, we maintain connectivity by contracting the incident arcs into a single arc, preserving connectedness. This operation will be called *reduction* to distinguish it from the simple removal of a vertex from a graph (Def. 5.14). Theorem 5.15 will then show that applying the reduction operation gives the join and split trees of the new, smaller graph.

Definition 5.13 *Define $T \ominus x_i$, the* reduction *of a C-tree T by a vertex x_i whose up-degree and down-degree are both ≤ 1, to be (see Fig. 5.4):*

1. *If x_i has arcs x_ix_j up and x_ix_k down in T, then: $T \ominus x_i = T \backslash x_i \cup x_jx_k$*
2. *Otherwise, $T \ominus x_i = T \backslash x_i$*

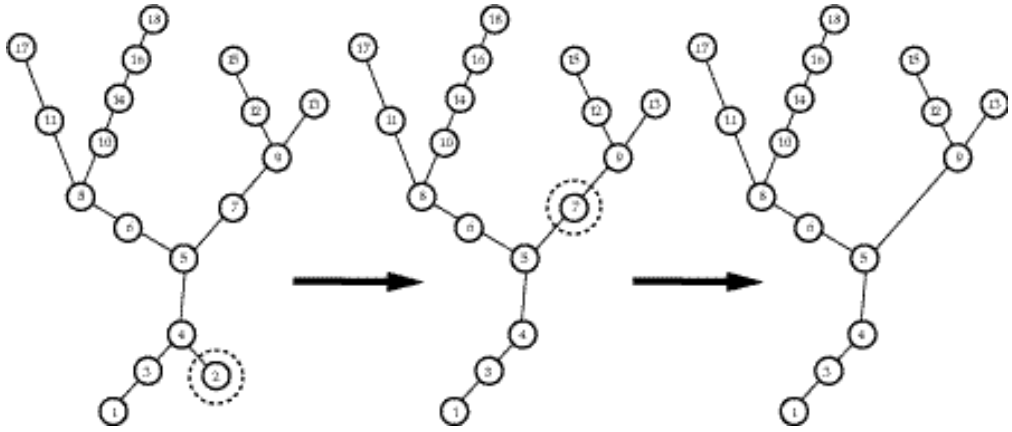


Figure 5.4: Reduction of a C-tree

Lemma 5.12 *Reduction does not increase the up- or down- degree of any vertex.*

Proof: This is easily observed, since the reduction deletes vertices and arcs, and only adds arc $x_j x_k$ when deleting $x_i x_j$ and $x_j x_k$.□

Definition 5.14 *The* subgraph of a graph G induced by removing a vertex x_i, *denoted* $G \backslash x_i$, *is defined to be the subgraph of* G *which includes all vertices of* G *except* x_i, *and all arcs of* G *except those incident to* x_i.

Lemma 5.13 *If* x_i *is a leaf of a C-tree* C, *and* $y_j y_k$ *is an arc of the corresponding join tree* J_C *such that* $h_j < h_k$, *and* $i \neq j, k$, *then* $y_j y_k$ *is also an arc of* $J_{C \backslash x_i}$.

Proof: For convenience, let C' refer to $C \backslash x_i$, and assume that paths have no repeated vertices.

Let P be any path in C. Since x_i is a leaf, this means that x_i cannot be on P (any path through x_i would have to duplicate an edge, and therefore

a vertex). Since x_i is not on P, the path P also exists in C'.

By Def. 5.5, x_j is adjacent to some vertex x_l in the component γ of $\Gamma_j^+(C)$ to which x_k belongs: i.e. there exists some path P from x_l to x_k in γ with no repeated vertices. But then P also exists in C', and therefore also exists in $\Gamma_j^+(C')$. Since $x_j x_l$ is also in C', x_j is adjacent to the component γ' of $\Gamma_j^+(C')$ to which x_k belongs.

Note that each path P connecting two vertices of γ will also be in γ', except for paths starting or ending at x_i: thus the vertices of γ are the same as those of γ', with the possible exception of x_i. It then follows that x_k is the smallest-valued vertex of γ', so by Def. 5.5, y_j is adjacent to y_k in $J_{C'}$.□

Corollary 5.14 *$J_C \ominus y_i$ is contained in $J_{C\backslash x_i}$.*

Proof: This follows from the observation that every arc of J_C that is not incident to y_i is also in $J_{C\backslash x_i}$.□

Theorem 5.15 *If x_i is a leaf of a C-tree C, then $J_{C\backslash x_i} = J_C \ominus y_i$.*

Proof: In Corollary 5.14, I showed that $J_{C\backslash x_i}$ contains all of the arcs of J_C that are not incident to y_i. I now show that the two are equivalent. For convenience, I will use C' to refer to $C\backslash x_i$. I separate the proof into three cases: x_i is an upper leaf, x_i is the global minimum, or x_i is a lower leaf (other than the global minimum):

Case I: x_i is an upper leaf:

If x_i is an upper leaf, it has a down-arc, and cannot be the global minimum x_1. By Lemma 5.6, y_i has one down-arc, say $y_i y_j$. Since J_C is a

tree with $n-1$ arcs (by Corollary 5.8), this leaves $n-2$ arcs of J_C that are not incident to y_i. But, by Lemma 5.13, each of these arcs must be in $J_{C'}$. Since $J_{C'}$ is a tree on $n-1$ vertices, it has $n-2$ arcs in total, and it follows that $J_C \backslash y_i = J_{C'}$. Since x_i is an upper leaf, y_i has no incident up-arc, and by Def. 5.13, part 2, $J_{C'} = J_C \backslash y_i = J_C \ominus y_i$.

Case II: x_i is the global minimum:

By hypothesis, x_i is a leaf, so $\delta^+(x_i) = 1$ and $\delta^-(x_i) = 0$. By Theorem 5.2, $\delta^+(y_i) = 1$, and by Lemma 5.6, $\delta^-(y_i) = 0$. Again, there are $n-2$ arcs of J_C that are not incident to y_i, each of which is also in $J_{C'}$, by Lemma 5.13, and $J_C \backslash y_i = J_{C'}$. Since x_i is the global minimum, y_i has no down-arc, and by Def. 5.13, part 2, $J_{C'} = J_C \backslash y_i = J_C \ominus y_i$.

Case III: x_i is a lower leaf other than the global minimum:

By hypothesis, x_i is a lower leaf, so $\delta^+(x_i) = 1$ and $\delta^-(x_i) = 0$. By Theorem 5.2, $\delta^+(y_i) = 1$, and by Lemma 5.6, $\delta^-(y_i) = 1$. This leaves $n-3$ arcs of J_C which are not incident to y_i, and by Lemma 5.13, each of them is also an arc of $J_{C'}$. Since $J_{C'}$ is a tree on $n-1$ vertices, it has $n-2$ arcs, so there is only one arc left unaccounted for.

Let the down-arc at y_i be $y_i y_j$, and the up-arc be $y_i y_k$ (see Fig. 5.5). I claim that $y_j y_k$ is an arc of $J_{C'}$. From Def. 5.5, x_i belongs to some component γ of $\Gamma_j^+(C)$, and x_j is adjacent to some vertex x_l in γ. Note that $x_l x_j$ must be a down-arc, and since x_i has no down-arcs, x_l cannot be x_i. Also, since x_i is the smallest-valued vertex in γ, $h_i < h_l$.

Consider the component ρ of $\Gamma_i^+(C)$ to which x_k belongs. Since each

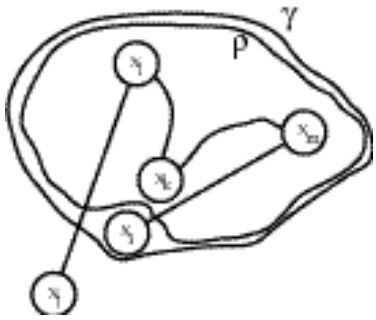


Figure 5.5: Reducing a Join Tree at a Lower Leaf

vertex of ρ has value $\geq h_k$, and $h_k > h_j$, $\rho \subseteq \gamma$, and x_k must be in γ. Since x_i is the smallest-valued vertex of γ, the only arc of γ that is not in ρ must be the arc incident to x_i, so $\rho = \gamma \backslash x_i$. But this must be a component of $\Gamma_i^+(C')$. Since $x_l \neq x_i$, it follows that x_l must have been connected to x_i in γ, as was x_k. Then x_l must be connected to x_k by a path whose vertices all have values $> h_i$. Therefore, x_l and x_k belong to the same component of ρ, and since x_k is the smallest-valued vertex of $\delta \backslash x_i$, y_j must be connected to y_k in $J_{C'}$.

Note that $y_j y_k$ cannot be an arc in J_C, because $y_i y_j y_k$ would then be a cycle in J_C, and J_C is known to be a tree by Corollary 5.8. Thus, I have added an arc to the $n-3$ arcs that we had already shown to be in $J_{C'}$, for a total of $n-2$. Since $J_{C'}$ is a tree on $n-1$ vertices, there are no more arcs to be found in $J_{C'}$.

From Corollary 5.14, $J_C \backslash y_i \subset J_{C'}$, I have just shown that $y_j y_k$ is the one remaining arc of $J_{C'}$., so $J_{C'} = J_C \backslash y_i \cup y_i y_j$. Since y_i had an up-arc $y_i y_k$ and a down-arc $y_i y_j$, it follows by Def. 5.13, part 1, that $J_{C'} = J_C \ominus y_i$.

Thus, in all three cases, $J_{C'} = J_C \ominus y_i$.$\square$

ReconstructCTree(join tree J_C, split tree S_C) returns C-tree C:

1. Base Case: If $\|J_C\| \leq 2$, let $C = J_C$.
2. Recursive Case:
   (a) Choose an index i such that $\delta^+(y_i) = 1$ and $\delta^-(z_i) = 0$ (a lower leaf), or $\delta^+(y_i) = 0$ and $\delta^-(z_i) = 1$ (an upper leaf).
   (b) If x_i is an upper leaf, find the incident arc y_iy_j in J_C: if x_i is a lower leaf, find the incident arc z_iz_j in S_C.
   (c) Let $J' = J_C \ominus y_i$, and $S' = S_C \ominus z_i$)
   (d) Let C' = ReconstructCTree(J', S')
   (e) Let $C = C' \cup x_ix_j$

Figure 5.6: Basic Reconstruction Algorithm

Dual 5.16 *If x_i is a leaf of a C-tree C, then $S_{C\setminus x_i} = S_C \ominus z_i$.*□

5.4 Basic Reconstruction Algorithm

In the previous sections, I have shown a number of properties of the join and split trees. I now put these together to obtain a recursive algorithm that reconstructs C from J_C and S_C:

Algorithm 5.1 *Algorithm to Reconstruct a C-Tree:*

In this algorithm (Fig. 5.6), recall the convention that the vertex x_i in the C-tree C corresponds to the vertex y_i in the join tree J_C and to the vertex z_i in the split tree S_C.

Theorem 5.17 *Given a join tree J_C and the corresponding split tree S_C for a C-tree C, Algorithm 5.1 correctly reconstructs the C-tree C.*

Proof: Proof is by induction on the size of J_C. Note that J_C, S_C, and C are all the same size.

Base case: $\|J_C\| \leq 2$.

Since C and J_C are C-trees, and there is only one possible C-tree on each of 0, 1, and 2 vertices, C is correctly computed by Step 1.

Inductive case: Assume that the algorithm works correctly for $\|J_C\| < k$, and show that it works correctly for $\|J_C\| = k$:

By Corollary 5.4, Step 2a correctly identifies a leaf of C from properties of J_C and S_C only.

By Lemma 5.10, Step 2b correctly identifies the arc of C that is incident to x_i.

By Theorem 5.15, Step 2c correctly constructs $J_{C\backslash x_i}$ and $S_{C\backslash x_i}$.

By the inductive hypothesis, since $\|J_{C\backslash x_i}\| = k-1$, ReconstructCTree() returns $C\backslash x_i$.

We have already observed that $x_i x_j$ is the arc of C that is incident to x_i. Thus, since C is a tree on k vertices, and $C' = C\backslash x_i$ is a tree on $k-1$ vertices, Step 2e correctly computes C. □

5.5 Improved Reconstruction Algorithm

A straightforward implementation of Algorithm 5.1 is quadratic in the size of the C-tree, but improvements are possible, based on the following observations:

1. Leaves in C other than the x_i chosen will remain leaves in $C\backslash x_i$ (from Lemma 5.12, and the observation that $C\backslash x_i$ is a tree).

2. After constructing J' and S', J_C and S_C are no longer needed

3. Before adding $x_i x_j$ to C', C is not needed.

4. The arcs $x_i x_j$ identified on different recursive levels are disjoint.

These observations allow elimination of tail-recursion, and result in a much more efficient algorithm, by making the following changes:

1. a queue containing the leaves of C is maintained at all times.

2. J' and S' are constructed by reducing J_C and S_C in place, rather than making copies.

3. C starts off empty, and is constructed in place, adding each $x_i x_j$ as it is identified

Algorithm 5.2 *Improved Algorithm to Reconstruct a C-Tree:*

In this algorithm (Fig. 5.7), I assume that the join tree J_C and split tree S_C are constructed using adjacency lists using half-arcs: that is, each arc $y_i y_j$ in J_C is stored as a directed arc α in y_i's adjacency list, linked to a directed arc α' in y_j's adjacency list.

Theorem 5.18 *Given a valid join tree J_C and the corresponding valid split tree S_C, Algorithm 5.2 correctly reconstructs the C-tree C.*

Proof: Since J_C and S_C are not used after constructing $J_{C\setminus x_i}$ and $S_{C\setminus x_i}$, reducing J_C and S_C in Step 3d, does not affect the correctness of the algorithm.

ImprovedReconstructCTree(join tree J_C, split tree S_C) returns C-tree C:

1. For each vertex x_i, if up-degree in J_C + down-degree in S_C is 1, queue x_i
2. Initialize C to an empty graph on $\|J_C\|$ vertices
3. While queue size > 1
   (a) Dequeue the first vertex on the leaf queue x_i
   (b) If x_i is an upper leaf, find incident arc y_iy_j in J_C. Otherwise, x_i is a lower leaf, so find incident arc z_iz_j in S_C.
   (c) Add x_ix_j to C.
   (d) $J_C \leftarrow J_C \ominus y_i$, $S_C \leftarrow S_C \ominus z_i$.
   (e) Test whether x_j is a leaf, and if so, transfer to leaf queue if it if not already on leaf queue.

Figure 5.7: Improved Reconstruction Algorithm

Similarly, since the arcs chosen at each iteration are disjoint, building C in place in Step 3c does not affect the correctness, either.

Correctness then depends upon the invariant that the leaf queue contains all vertices of C' at all times, where C' is the reduced C-tree that corresponds to the reduced join and split trees. Initially, all vertices are scanned, and the leaves placed on the queue. From Lemma 5.12, the degree of vertices in J_C and S_C never increase as reductions take place. Since the up-degree of a vertex in C' matches the up-degree in J_C, and the down-degree matches the down-degree in S_C, it follows that the degrees in C' never increase. Also, since C' must remain a tree at all times, the degree of a leaf on the queue cannot decrease (this would result in a vertex with degree 0, disconnecting the tree).

From the mechanics of reduction in the join tree, either an upper leaf, the global minimum, or a lower leaf is reduced.

Case I: x_i is an upper leaf:

Theorem 5.15 showed that y_i is adjacent to precisely one vertex y_j. It then follows that y_j has its degree reduced by 1, and that all other vertices retain their degree after the reduction. In this case, Step 3e maintains the property as required.

Case II: x_i is a lower leaf other than the global minimum:

If x_i was a lower leaf other than the global minimum, Theorem 5.15 shows that y_i is between y_j and y_k: we replace y_iy_j and y_iy_k with y_jy_k: in this case, neither y_j nor y_k has its degree reduced, so y_j does not become a leaf, unless it already was, and neither does y_k. Again, Step 3e maintains the property as required.

Case III: x_i is the global minimum:

If x_i is the global minimum, z_i is adjacent to some z_j, and by Corollary 5.11, x_i is adjacent to x_j. Since x_i is the global minimum, y_i has no down arcs, and must have at least one up arc. But since x_i is a leaf, $\delta^+(x_i) = \delta^+(y_i) = 1$. Let the up arc from y_i be y_iy_k. If y_k is y_j, then deleting y_i from J_C makes y_j the new global minimum, and does not change the degree of any other vertex of J_C. It then follows that Step 3e maintains the property as required.

Otherwise, y_j is not the same as y_k. By Def. 5.5, x_k is the smallest-valued vertex of some component γ in $\Gamma_i^+(C)$. Also, because x_i is a leaf,

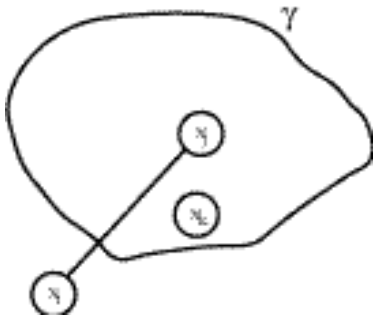


Figure 5.8: The Leaf Queue and the Global Minimum

and adjacent to x_j, x_i cannot be adjacent to x_k. Thus, since x_i is the global minimum, x_k is not adjacent to any x_m in C such that $h_m < h_k$, which is to say that $\delta^-(x_k) = 0$ and $\delta^-(z_k) = 0$ before the reduction takes place.

Since $h_i < h_k$, deleting the arc $y_i y_k$ cannot affect $\delta^+(y_k)$, and by Theorem 5.2, $\delta^+(x_k)$ is also unaffected. I now break into subcases depending on the up-degree of x_k (which is the same as the up-degree of y_k):

Subcase IIIa: $\delta^+(y_k) = 1$**:**

Since $\delta^-(z_k) = 0$, x_k was a lower leaf before the reduction took place, by Corollary 5.5. Thus, by the invariant, x_k was already on the leaf queue, and does not need to be tested at this iteration.

Subcase IIIb: $\delta^+(y_k) > 1$**:**

If $\delta^+(y_k) > 1$ before reduction, $\delta^+(y_k) > 1$ after reduction, and x_k does not need to be tested at this iteration. Step 3e maintains the property as required.

Subcase IIIc: $\delta^+(y_k) = 0$**:**

This case can never happen. To see this, observe that J_C is a tree (Corollary 5.8), and $\delta^-(y_k) = 1$ before reduction (Lemma 5.6). After deleting

y_iy_k from J_C, $\delta^+(y_k) = \delta^-(y_k) = 0$, which is only possible if J_C has size 1. But in this case, there can be no other vertex y_j in J_C, which contradicts the assumption that x_i was connected to x_j in C. Thus it follows that $\delta^+(y_k) \neq 0$ before reduction.

I have now shown that, in all cases that can occur, Step 3e maintains the invariant.□

Theorem 5.19 *Algorithm 5.2 takes $O(t)$ time and space to reconstruct a C-tree C with t vertices.*

Proof: Again, we consider the time and space requirements for each step:

1. Step 1 takes $O(1)$ time to test each vertex to see if it is a leaf. Since a leaf has maximum degree 2 in either the join or split tree, it takes $O(1)$ time to test each vertex, for a total of $O(t)$ time to check all vertices.

2. Step 2 takes at most $O(t)$ time to initialize C.

3. Step 3 iterates $t-1$ times, since one arc is added to C on each iteration.

   (a) Step 3a dequeues the first vertex on the leaf queue x_i in $O(1)$ time

   (b) Step 3b takes $O(1)$ time to locate the arc in J_C or S_C.

   (c) Step 3c takes $O(1)$ time to add the arc to C.

   (d) Step 3d takes $O(1)$ time to reduce J_C and S_C. Since x_i is a leaf, y_i has maximum degree of 2, and the arcs y_iy_j and y_iy_k can be located in $O(1)$ time each for deletion. Since the arcs are stored

using half-arcs in adjacency lists have half-arcs at each vertex, $y_j y_k$ can be added as follows: Locate the half-arcs α and β from y_i to y_j and y_k. For each of these arcs, find the corresponding half-arc α' and β'. Delete α and β, and connect α' to β'. This takes at most 3 operations to reduce each tree, each of which is performed in $O(1)$ time.

(e) Step 3e tests whether x_j is a leaf in $O(1)$ time, and if so, transfers it to the leaf queue in $O(1)$ time.

Summing up the time required for the algorithm gives a total of $O(t)$ time. $O(t)$ space is required to store each of J_C, S_C, and C, since they are trees on t vertices. The leaf queue will also require $O(t)$ space, for a total of $O(t)$ space required.□

5.6 Join Trees of the Mesh & the Contour Tree

The algorithm described in the previous section is useful only if we have a convenient way of obtaining the join and split trees for the contour tree. Fortunately, the join and split trees for the augmented contour tree are identical to those for the mesh. To show this, I start with the trivial observation that the contour tree is a C-tree: this implies that the join tree is well-defined, and that the algorithm in the previous section can reconstruct the contour tree correctly.

Lemma 5.20 *The contour tree C of a mesh M is a C-tree.*

Proof: Def. 5.1 defines a height graph to be a graph whose vertices have heights associated, in sorted order. We know from Def. 4.16, p.36 that the vertices of the contour tree are finite contour classes, and are thus associated with the vertices $x_1, \ldots, x_n$, which are assumed to be in sorted order by Assn. 4.1, p.30. Thus a contour tree is a height graph.

Def. 5.2 defines a C-tree to be a height graph that is also a tree. By Lemma 4.3, p.38, we know that the contour tree is a tree when the mesh is connected. Since this is the case that interests us, it then follows that the contour tree is a C-tree.□

In Def. 4.16, p.36, I defined the contour tree to be a tree whose vertices were the critical points of the mesh. In Def. 5.5, I defined the join tree so that it included all the vertices of the mesh. Clearly, these two vertex sets need not be the same. However, I also defined the augmented contour tree in Sec. 4.7, p.41 in such a way that it included all the vertices of the mesh. Thus, if I can relate the join tree of the mesh to that of the augmented contour tree, it only remains to reduce the augmented contour tree to the contour tree.

I will show that the components of the subgraphs above [below] any vertex in the augmented contour tree are essentially identical to the corresponding components of the mesh, but first, I must prove a preliminary result:

Lemma 5.21 *x_i and x_j belong to the same component of $\Gamma_k^+(M)$ precisely when x_i and x_j belong to the same component of $\{x : f(x) > h_k\}$.*

Proof:

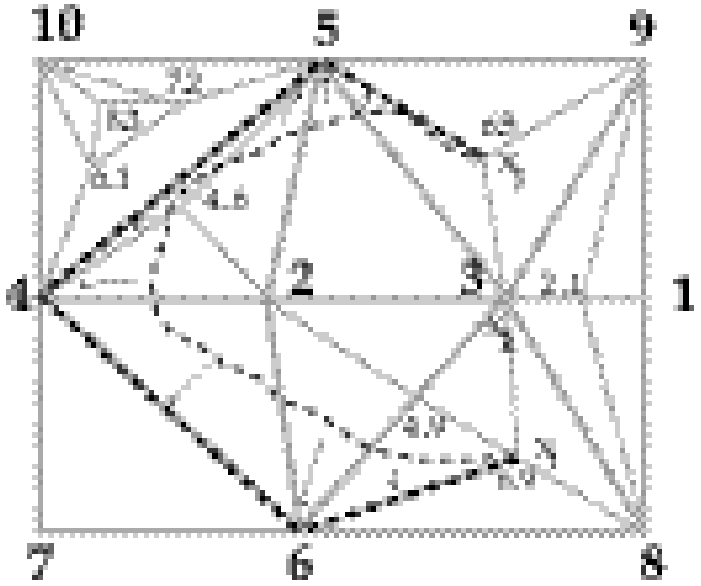


Figure 5.9: Constructing a graph path from a path in space

($\Rightarrow$): It suffices to show that this is true for any x_i and x_j connected in the mesh M by the edge $x_i x_j$. From Assn. 1.4, p.7, the interpolation is piecewise-linear over the cells of the mesh. Any point x on the edge $x_i x_j$ must have a value between those of x_i and x_j: thus x belongs to $\{x : f(x) > h_k\}$. Since this is true for all points along the edge, x_i and x_j must belong to the same connected component of $\{x : f(x) > h_k\}$.

($\Leftarrow$): Let x_i and x_j be connected in $\{x : f(x) > h_k\}$. Then there exists some path P from x_i to x_j such that $f(p) > h_k$ for all points p in P (see Fig. 5.9). Since the mesh is assumed to be simplicial (Assn. 1.3, p.7), the path P is completely contained by the sequence of simplices through which it passes. If a vertex of one of these simplices has value $> h_k$, then the vertex belongs to the same component of $\{x : f(x) > h_k\}$ as P (by Assn. 1.4, p.7).

Similarly, any edge between two vertices of a simplex on the path is in the same component if both ends of the edge have value $> h_k$. P may only enter and exit a simplex through faces which have at least one vertex higher than x_k. Pick the highest vertices x_p and x_q on the entry and exit faces. Since these vertices are in the same simplex, they are adjacent, and the edge $x_p x_q$ is in $\{x : f(x) > h_k\}$. Thus, I can replace the path P with a path Q which travels only along edges of the mesh: this path Q must also exist in $\Gamma_k^+(M)$, since it is composed of edges between vertices with values $> h_k$. It then follows that x_i and x_j belong to the same connected component of $\Gamma_k^+(M)$.□

Dual 5.22 *x_i and x_j belong to the same component of $\Gamma_k^-(M)$ precisely when x_i and x_j belong to the same component of $\{x : f(x) < h_k\}$.*□

Lemma 5.23 *For each component in $\Gamma_i^+(AC_M)$ (where AC_M is the augmented contour tree for the mesh M), there exists a component in $\Gamma_i^+(M)$ containing exactly the same vertices.*

Proof: Proof is by finite induction, starting with the highest vertex x_n, for which the property is trivially true.

Assuming that the hypothesis is true for $k \leq i \leq n$, I now consider the vertex x_{k-1}: the only difference between the components of $\Gamma_k^+(M)$ and $\Gamma_{k-1}^+(M)$ is that the up-arcs from x_k have been added to the latter. Thus, I break the proof into three cases: local maxima, joins, and other points:

Case I: x_k is a local maximum:

Since x_k has no up-arcs, there are no edges added to $\Gamma_k^+(M)$ to obtain $\Gamma_{k-1}^+(M)$. What is the corresponding change in the augmented contour tree?

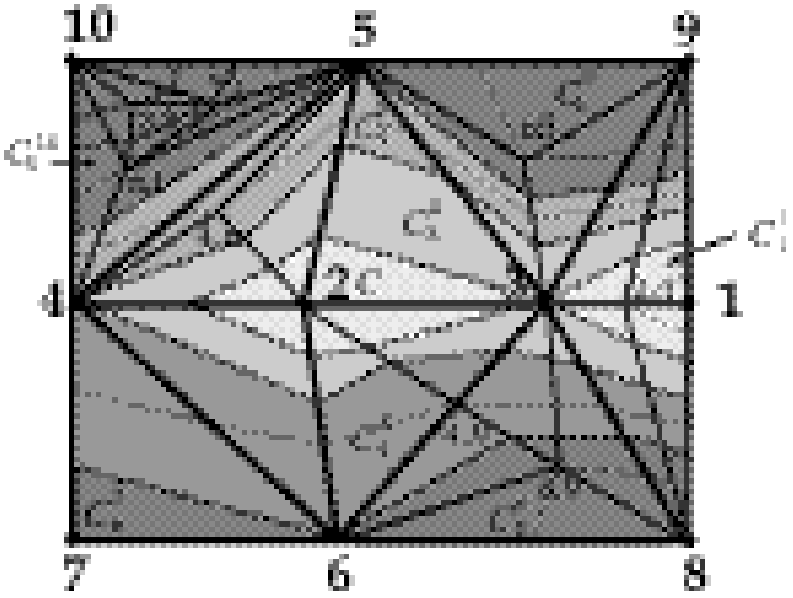


Figure 5.10: Region Representing an Edge in Augmented Contour Tree

From Def. 4.16, p.36, a local maximum only has one edge: this descends from it. Thus, since x_k is not adjacent to any components of $\Gamma_k^+(M)$, no edges are added to $\Gamma_k^+(AC_M)$ to obtain $\Gamma_{k-1}^+(AC_M)$. By the inductive hypothesis, the components of $\Gamma_k^+(M)$ and $\Gamma_k^+(AC_M)$ contain the same vertices. It then follows that the components of $\Gamma_{k-1}^+(M)$ and $\Gamma_{k-1}^+(AC_M)$ contain the same vertices.

Case II: x_k is a join:

Suppose that x_k is adjacent to x_j in AC_M, with $h_k < h_j$. From Def. 4.17, p.41, x_k and x_j both either belong to some superarc, or are endpoints of it. Since the superarcs and supernodes correspond to connected contour classes, I take the union of these contour classes, and obtain a connected set in the original space of points with values between h_k and h_j. Therefore, there is a path P from x_k to x_j in this set (see Fig. 5.10).

But this set is contained in some component γ of $\{x : f(x) > h_{k-1}\}$. So, by Lemma 5.21, x_k and x_j must also be connected in $\Gamma^+_{k-1}(M)$. This is true for each edge $x_k x_j$ in AC_M (with $h_k < h_j$). Also, the components of $\Gamma^+_k(M)$ and $\{x : f(x) > h_k\}$ have the same vertex sets by the induction hypothesis. Thus, it follows that x_k is connected to the same components of $\Gamma^+_k(M)$ in M as in AC_M.

As a result, the component of $\Gamma^+_{k-1}(M)$ to which x_k belongs will correspond directly to the component of $\{x : f(x) > h_{k-1}\}$ to which x_k belongs. Components to which x_k does not connect will be unaffected, so we conclude that the components of $\Gamma^+_{k-1}(M)$ and $\Gamma^+_{k-1}(AC_M)$ contain the same vertices, as required.

Case III: x_k is neither a join nor a local maximum:

In this case, x_k is adjacent to only one component of $\Gamma^+_k(M)$, and an argument similar to that of Case II applies to show that the components of $\Gamma^+_{k-1}(M)$ and $\Gamma^+_{k-1}(AC_M)$ contain the same vertices.□

Dual 5.24 *For each component in $\Gamma^-_i(AC_M)$, there exists a component in $\Gamma^-_i(M)$ containing exactly the same vertices.*□

A corollary of this result is that the augmented contour tree has the same join tree as the mesh does:

Corollary 5.25 *The augmented contour tree AC_M and the mesh M have the same join tree (i.e. $J_{AC_M} = J_M$).*

Proof: In Def. 5.5, I defined the join tree of a height graph G in terms of the components of $\Gamma^+_i(G)$. By Lemma 5.23, these components are identical in

AC_M and M, and we saw in the proof of the theorem that x_i will be connected to the same components of $\Gamma_i^+(AC_M)$ and $\Gamma_i^+(M)$. It follows immediately from Def. 5.5 that $J_{AC_M} = J_M$.□

Dual 5.26 *The augmented contour tree AC_M and the mesh M have the same split tree (i.e. $S_{AC_M} = S_M$).*□

Chapter 6

Join & Split Trees of the Mesh

The contour tree is useful because it represents the topology of the level sets in the data. Taking a cross-section of the contour tree at a given height h gives us the connectivity of the level set $\{x : f(x) = h\}$.

Corresponding to this property, it can be shown that the join tree and split tree of the mesh give us the connectivity of the sets $\{x : f(x) \geq h\}$ and $\{x : f(x) \leq h\}$ respectively: the *interior* and *exterior* of the level set. Describing these sets as "interior" and "exterior" is consistent with sweeping through the isovalues from high to low: the interior is the set of points enclosed by the level set as it expands "outwards."

This property, although illustrative, is not necessary for what follows: an algorithm for the construction of the join and split trees of a sorted simplicial mesh in $O(N\alpha(N))$ time and $O(n)$ space. The following chapter uses this algorithm, and the algorithms demonstrated in the previous chapter, to construct the contour tree for the mesh.

Throughout this chapter, I shall use M to refer to the simplicial mesh that fills the spatial volume. Since each vertex x_i of M has an associated height h_i, in sorted order, M satisfies the definition of a height graph in Def. 5.1, p.47.

6.1 Construction of the Join Tree

From Def. 5.5, p.48, each vertex y_i of J_M must be connected to the smallest-valued vertex in each component in $\Gamma_i^+(M)$. At y_i, these components will merge: at lower values of i, there will be one component corresponding to all of them. This property makes it possible to use Tarjan's discrete set union algorithm [24] to compute J_M.

Tarjan's algorithm progressively constructs the connectivity of subgraphs of M by adding edges to the union-find structure, in any arbitrary order. At each step, the union-find structure represents the connectivity of the subgraph of M induced by the edges already added.

Each edge $x_j x_i$ in M runs from a "higher" end x_j to a "lower" end x_i (i.e. $h_j > h_i$, or $j > i$). Since Tarjan's algorithm does not require any particular order, I add the edges of M to the union-find structure in order of the height of the lower end. This leads to the following algorithm:

Algorithm 6.1 *Algorithm To Construct J_M:*

In the algorithm (Fig. 6.1), the only item added to the usual union-find representation is LowestVertex, which stores the lowest vertex for each component: this allows me to add edges to the join tree at each step.

1. for i := n downto 1 do:
   (a) Component[i] := NONE
   (b) for each vertex x_j adjacent to x_i
      (i). if ($j < i$) or (Component[i] = Component[j]) skip x_j
      (ii). if Component[i] = NONE
         (A). Component[i] := Component[j]
              else
         (B). UFMerge(Component[i], Component[j])
      (iii). AddEdgeToJoinTree(y_i, LowestVertex[Component[j]])
      (iv). LowestVertex[Component[j]] := y_i
   (c) if Component[i] = NONE
      (i). Component[i] := i
      (ii). LowestVertex[i] := y_i

Figure 6.1: Algorithm to Construct Join Tree

To show that this algorithm correctly computes J_M, I shall prove by induction that the components in the union-find structure correspond directly with the components of $\Gamma_i^+(M)$ immediately before iteration i. Next I show that LowestVertex holds the lowest vertex in each component. Finally, I shall show that Step 1(b)(iii) generates all the edges of J_M.

Lemma 6.1 *Immediately before step i, the components of the union-find structure used by Algorithm 6.1 correspond to the components of $\Gamma_i^+(M)$.*

Proof: Proof is by finite induction, with base case of $i = n$. In this base case, the union-find structure is empty, and $\Gamma_n^+(M)$ contains no vertices. Since neither has any components, the correspondence is obvious.

I assume, therefore, that the result is true for $k \leq i \leq n$, and show that

the result holds for $k-1$. The proof now breaks into three cases, depending on whether x_k is adjacent to 0, 1, or ≥ 2 components of $\Gamma_k^+(M)$.

Case I: x_k has no components adjacent yet:

Since vertices x_n to x_{k+1} have already been processed, this means that x_k is not adjacent to any higher vertices. Thus x_k is a local maximum, and $j < k$ for every vertex x_j adjacent to x_k. Accordingly, Step 1(b)(i) causes the algorithm to skip each of these vertices, and when Step 1b completes, Step 1c executes, setting x_k to represent a new component in the union-find structure. This corresponds to the component $\{x_k\}$ in $\Gamma_{k-1}^+(M)$. Any other component in $\Gamma_{k-1}^+(M)$ must consist solely of vertices higher than x_k. By the inductive hypothesis, these were correctly represented prior to step k. Since step k does not merge any of these components, it follows that these components are still correctly represented in the union-find structure.

Case II: All adjacent vertices higher than x_k belong to one component:

In this case, x_k is adjacent to vertices of exactly one component in $\Gamma_k^+(M)$. When the first of these vertices is encountered, Step 1(b)(ii)(A) adds x_k to the component. Subsequent neighbours higher than x_k all belong to this same component, and are skipped due to Step 1(b)(i). Thus, x_k is added to the component, and all other components are left untouched.

Case III: Adjacent vertices higher than x_k belong to more than one component:

As in Case II, Step 1(b)(ii)(A) adds x_k to the first component encoun-

tered adjacent to x_k. When the first vertex from a different component is encountered, Step 1(b)(ii)(B) merges the two components together: subsequent adjacent vertices belonging to either of these components will then be skipped at Step 1(b)(i). Thus, the first vertex from each adjacent component causes that component to merge onto x_k's component. As in the previous cases, components not adjacent to x_k are unaffected.

The result then follows by induction.□

Lemma 6.2 *For each component γ in $\Gamma_i^+(M)$, LowestVertex[γ] holds the smallest vertex in γ before step i of Algorithm 6.1.*

Proof: Again, proof is by finite induction on i. The base case is trivial: no components exist before step n. In Case I of Lemma 6.1, I showed that Step 1(b)(ii) is never reached: only Step 1c is executed, creating a component containing only x_k. Step 1(c)(ii) then sets it's lowest vertex correctly to x_k.

In Case II of Lemma 6.1, I showed that Step 1(b)(ii)(A) adds x_k to an existing component. Step 1(b)(iv) then sets the lowest vertex of this component to x_k. Since the only vertices in this component prior to step k were higher than x_k, x_k is in fact the lowest vertex: other components are unaffected.

In Case III of Lemma 6.1, I showed that all components adjacent to x_k were merged by Step 1(b)(ii)(B): Step 1(b)(iv) then sets the lowest vertex correctly to x_k.

The result again follows by induction.□

Theorem 6.3 *Algorithm 6.1 correctly computes J_M.*

Proof: The two preceding lemmas (Lemma 6.1 and Lemma 6.2) have established that the union-find represents the components of $\Gamma_i^+(M)$ at step i, and that LowestVertex holds the smallest-valued vertex of each component at the same time.

From Def. 5.5, p.48, we see that y_i is connected to y_j in J_M precisely when x_j is the smallest-valued vertex of a component γ of $\Gamma_i^+(M)$, and x_i is adjacent to some vertex in γ. But this is exactly the edge we add to the join tree in Step 1(b)(iii). It follows that the result is J_M.□

Theorem 6.4 *Algorithm 6.1 computes J_M in $O(N + t\alpha(t))$ time and $O(n)$ space.*

Proof: The steps of the algorithm each execute the following number of times:

1. Step 1 executes n times
2. Step 1a executes once per loop, for a total of n times
3. Step 1b executes once for each incident edge at each vertex, or twice for each edge in the mesh in total. As noted in Def. 2.8, p.12, the total number of edges in a simplicial mesh is $O(N)$.
4. Step 1(b)(i) executes once per edge (at the lower end): a total of $O(N)$ times.
5. Step 1(b)(ii) executes at most once per vertex: a total of $O(n)$ times, with $O(1)$ work on each execution

6. Step 1(b)(ii)(B) performs at most t merges, for a total time of $O(t\alpha(t))$

7. Step 1c executes once for each local maximum, for a total of $O(t)$ times.

Summing these times up, we get a total of $O(N + n + t\alpha(t))$, which simplifies to $O(N + t\alpha(t))$. Space requirements are $O(n)$ for the union-find structure, $O(n)$ for the lowest vertex, and $O(n)$ for the join tree that is created, for a total of $O(n)$ space.□

Dual 6.5 *By the same duality observed in Dual 5.1, p.49, the split tree S_M may be computed in $O(N + t\alpha(t))$ time and $O(n)$ space.*□

Chapter 7

Contour Tree Construction Algorithm

In the previous chapters, I have defined the contour tree (Ch. 4), described how to reconstruct a C-tree from join and split trees (Ch. 5), and shown how to compute the join and split trees of the mesh efficiently (Ch. 6). In this chapter, I put these pieces together to get an efficient algorithm for computing the contour tree of the entire mesh.

7.1 Constructing the Augmented Contour Tree

In Sec. 5.6, p.68, I showed that the mesh and the augmented contour tree have identical join and split trees. From this, Algorithm 6.1, p.76 and Algorithm 5.2, p.63, I can now assemble an algorithm to compute the augmented contour tree:

Algorithm 7.1 *Algorithm to construct the Augmented Contour Tree*

Given a mesh M, compute the augmented contour tree AC_M as follows:

1. Use Algorithm 6.1, p.76 to compute the join and split trees J_M and S_M.

2. Use Algorithm 5.2, p.63 to compute AC_M from J_M and S_M.

Theorem 7.1 *Algorithm 7.1 correctly constructs the augmented contour tree for the mesh M.*

Proof: By Theorem 6.3, p.79, Step 1 correctly computes J_M and S_M. By Corollary 5.25 and Dual 5.26, these are identical to J_{AC_M} and S_{AC_M}. And by Theorem 5.18, p.63, Step 2 correctly computes AC_M.□

Theorem 7.2 *Algorithm 7.1 computes the augmented contour tree for the mesh M in $O(N + t\alpha(t))$ time and $O(n)$ working space.*

Proof: By Theorem 6.4, p.80, Step 1 takes $O(N + t\alpha(t))$ time and $O(n)$ space. By Theorem 5.19, p.67, Step 2 takes $O(n)$ time and space. Since $n = O(N)$, the result follows.□

7.2 Constructing the Contour Tree

I have now given an algorithm for computing the augmented contour tree. But what of the contour tree itself? Fortunately, it is easy to compute the contour tree from the augmented contour tree by using the reduction operation defined in Def. 5.13, p.56. This leads to the following algorithm for computing the contour tree:

Algorithm 7.2 *Algorithm to construct the Contour Tree*

Given a mesh M, we compute the contour tree C as follows:

1. Use Algorithm 7.1 to compute the augmented contour tree AC
2. For each vertex x_i in AC
   (a) If $\delta^+(i) = \delta^-(i) = 1$
      (i). Reduce vertex x_i

Theorem 7.3 *Algorithm 7.2 correctly constructs the contour tree for the mesh M.*

Proof: By Theorem 7.1, p.83, Step 1 correctly computes the augmented contour tree AC. In Sec. 4.7, p.41, I observed that the ordinary points of the mesh were inserted into the superarcs to which they belonged, and that each ordinary point has one up-arc, and one down-arc. This is not true for critical points, since these will always have either the up-degree or down-degree $\neq 1$

Thus, Step 2a correctly identifies the ordinary points in the contour tree, and Step 2(a)(i) correctly removes them from the contour tree.□

Theorem 7.4 *Algorithm 7.2 computes the contour tree for the mesh M in $O(N + t\alpha(t))$ time and $O(n)$ working space, and requires $O(t)$ output size.*

Proof: By Theorem 7.2, Step 1 requires $O(N + t\alpha(t))$ time and $O(n)$ working space. Step 2 executes $O(n)$ times, but, as in Theorem 5.19, p.67, Step 2a and Step 2(a)(i) take $O(1)$ time for each vertex. This gives a total of

$O(N + t\alpha(t))$ time and $O(n)$ working space. The output is the contour tree, which requires $O(t)$ space. □

Although Algorithm 7.2 correctly computes the contour tree, note that it assumes that the vertices $x_1, \ldots, x_n$ are in sorted order. If this is not the case, then the cost of sorting must be included:

Corollary 7.5 *Algorithm 7.2 computes the contour tree for an unsorted mesh in* $O(n \log n + N + t\alpha(t))$ *time and* $O(n)$ *working space.*□

Finally, some small optimizations of this algorithm are possible. If the augmented contour tree is not required, it is possible to use the reduction operation to suppress ordinary points during the computation of the join and split trees (see Algorithm 6.1, p.76). In this case, we store the HighestVertex for each component as well as the LowestVertex, and only generate a join tree arc at a join or the global minimum. Note that splits and local minima will not be represented in this join tree: they must be inserted.

Fortunately, all splits and local minima are identified in the split tree. If sufficient information is retained from the join tree computation, it is easy to determine which arc of the join tree the splits and local minima belong to. Inserting these vertices on this arc in sorted order then gives the correct join tree for the contour tree. Since the vertices are already sorted, this can be done in $O(1)$ time per vertex, for a maximum of $O(t)$ time. However, doing this adds a significant amount of complexity for relatively little gain, since the augmented contour tree is used for the purpose of generating seed sets (see next chapter).

It appears to be possible to avoid the $O(n \log n)$ cost for sorting the vertices. To do this, note that, for each vertex x_i, we need to know the components in the union-find of it's neighbours only: this does not depend on a global ordering of the vertices. If components are "grown" downwards from each local maximum, it should be possible to avoid sorting entirely. However, this would require separate search queues for each local maximum: this makes the algorithm much more complex, and is likely to be prohibitive in practice.

Note that the $O(n \log n)$ in this algorithm is entirely due to the sort, and not due to the cost of building a data structure. Since sorting can usually be done efficiently in practice, the cost of sorting may not be a major concern.

Chapter 8

Generating Seeds

The contour tree represents the connectivity of all level sets: to generate the isosurfaces, we need seeds for contour-following (Sec. 3.5.1). I start by computing both the contour tree and the augmented contour tree. The contour tree is used to identify which superarcs intersect the desired level set: this can be done by searching all superarcs, or by storing the superarcs in an interval tree. Once the relevant superarcs are identified, a seed edge is selected for each superarc. This can be done in one of several ways:

8.1 Simple Seed Sets

The simplest method is to store the entire augmented contour tree. To identify a seed cell for an isovalue h, the corresponding superarcs are searched in the augmented contour tree to find the arc $x_i x_j$ for which $h_i < h < h_j$. However, x_i and x_j need not be adjacent to each other in the mesh. If x_i is not a split,

all its down arcs intersect the desired contour, so we simply pick a down-arc. Similarly, if x_j is not a join, any of its up arcs can be used as a seed. If x_i is a split and x_j is a join, a problem arises: it is difficult to pick an edge for the interpolation. This can be dealt with by picking an up arc for each up superarc when we construct the join tree, and a down arc for each down superarc, although this adds complexity.

This simple search takes $O(n)$ time, if done with a linear search, or $O(\log n)$ for a binary search. In extreme cases, this could lead to $\Theta(n \log n)$ cost to find the seed set.

8.2 Heuristic Seed Sets

Instead of precomputing the seed set, I borrow from Itoh & Koyamada [16, 15] and Bajaj et al. [2] the trick of associating a path with each superarc. This path can be computed in the following way at runtime:

1. At a local maximum, start the path with the edge to the lowest-valued adjacent vertex.

2. At a local minimum, start the path with the edge to the highest-valued adjacent vertex.

3. At a join, each ascending superarc corresponds to a join component that terminates at the join. On each such superarc, store the edge between the current vertex, and the highest adjacent vertex belonging to that join component. Start from this edge when finding a seed.

4. At a split, each descending superarc corresponds to a split component: again, store a starting edge on the superarc.

5. If none of the above apply, the superarc must descend from a join to a split: in this case, there is only one descending superarc from the join (and only one ascending superarc from the split). Therefore, any edge to a lower-valued vertex adjacent to the join vertex can be used to start the path, as can any edge to a higher-valued vertex adjacent to the split vertex.

In all these cases, a supernode has been identified, as has an adjacent starting node on either an ascending or descending path. Assuming an ascending path, if the supernodes do not already straddle the desired isovalue, advance to the edge from the adjacent node to it's highest adjacent neighbour. Repeat until the supernodes straddle the desired isovalue.

Note that this heuristic will never reach a dead end at a local maximum: if there is more than one maximum to be found in this way, there must be a join vertex at a lower value than all such maxima. The desired superarc must then terminate at or before the join vertex.

If the paths are to be preprocessed and stored, perform a breadth-first search in the same fashion for the isovalue at the far end of the superarc, thus finding the shortest path to a vertex above that isovalue. This preprocessing can be performed in $O(N)$ time and $O(n)$ space, since the same edge is never considered more than once.

As noted in [17], a dataset can consist entirely of local extrema: in this case, $t = \Omega(n)$. In this case, it will take $\Theta(n)$ time to generate a level set that intersects all of the superarcs. But no algorithm can improve on this, since the output size k will be $\Theta(n)$.

To determine which superarcs span a level set, the superarcs can be stored in an interval tree, as Cignoni et al. [5] do with the edges of the original dataset. This is used by Bajaj etal in [2]. Using this approach reduces the time to generate a level set to $O(\log t) + k$, at the cost of $O(t \log t)$ preprocessing. Since we expect that $t \ll N$ (Def. 2.10, p.13), storing superarcs in this way should not be necessary for crystallographic data.

Chapter 9

Implementation

This chapter discusses the major issues that arose during the implementation of the algorithm, and the solutions adopted in each case.

9.1 Simplicial Subdivision

The first difficulty to be resolved is the assumption that the data is acquired upon a rectilinear grid(Assn. 1.2). Ideally, we would like to use the rectilinear grid directly, with the natural tri-linear interpolation function over each voxel.

Unfortunately, all existing contour tree algorithms assume that the data is in the form of a simplicial mesh: the simplices prevent ambiguities of the interpolating function inside the mesh(Assn. 1.3).

The immediate consequence of this disparity is that either the algorithm must be modified so that it works with voxels, or the grid must be converted into a simplicial mesh. I chose to convert the grid to a simplicial mesh, by

subdividing each voxel into simplices.

9.1.1 Desiderata for Subdivision

In order for the algorithm to work, our definition of a mesh (Def. 2.1, p.9) requires that faces be shared between adjacent cells. Thus, where the face of a voxel is subdivided, the subfaces must be the same in both of the voxels that intersect at that face.

Note that, in choosing a subdivision, we implicitly choose an interpolation function over the original voxel, composed of the interpolation function for each simplex in the voxel.

In choosing a subdivision, the ideal is to approximate the tri-linear interpolation function over the voxels. This gives several criteria for quality, to which we add some criteria related to efficiency of processing. Not surprisingly, it is not possible to satisfy all of the following goals:

ii) the interpolation function for a given point should depend solely on the values at the vertices of the voxel containing the point.

ii) the subdivision should be symmetrical: all vertices should be treated equally in a given cell.

iii) the subdivision should not magnify the dataset - i.e. it should not require the addition of data points.

iv) the subdivision should generate as few simplices as possible.

v) if possible, the subdivision should be implicit, for processing efficiency.

(a) The Subdivision (b) Even Parity (c) Odd Parity

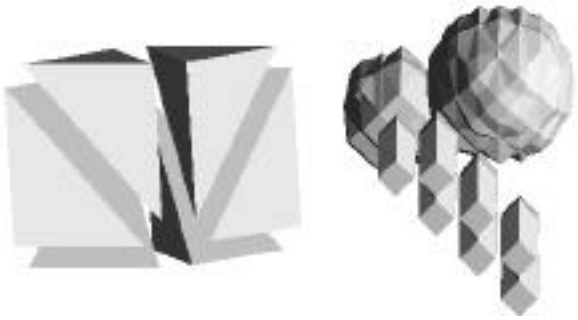
Figure 9.1: Minimal subdivision: 5 simplices / voxel

9.1.2 Some Possible Subdivision Schemes

A number of schemes for subdividing a voxel have been used, or could potentially be used. One of the sample datasets, "atom9" is used to demonstrate deviations from the ideal tri-linear interpolation. This dataset was constructed artificially, by placing local "atoms" in $\mathbb{R}^3$. Each "atom" was assumed to be the centre of a Gaussian distribution of electron density: the volume was sampled at regular intervals to produce the data. One area of this dataset proved especially good at revealing artefacts due to the subdivision: a zig-zag of atoms placed on vertices of the sampling grid.

a) Minimal subdivision, using 5 simplices (Fig. 9.1(a)).

   This fails criterion ii): not all vertices are treated equally. Fig. 9.1(b) and Fig. 9.1(c) shows the result of applying this subdivision to a sample dataset. This asymmetry could be mitigated by randomizing the orientation of the subdivision in each voxel, which would violate our condition



(a) The Subdivision (b) Artefacts

Figure 9.2: Axis-aligned subdivision: 6 simplices / voxel

that subfaces must match between voxels.

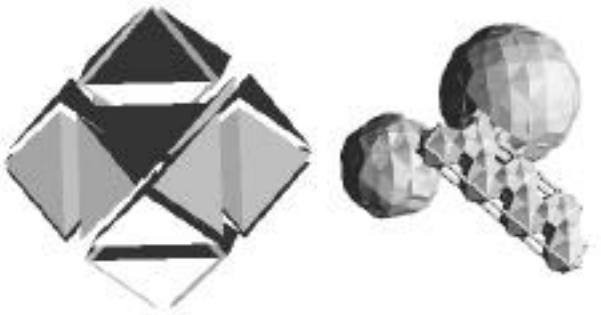
b) Axis-aligned subdivision, with 6 simplices, arranged around a major diagonal of the voxel (Fig. 9.2(a)).

   Again, this fails condition ii). Fig. 9.2(b) shows the result of applying this subdivision to a sample dataset. As with a), this asymmetry could be reduced by randomized orientations, but the same difficulty would be encountered; it would violate our subface-matching condition.

c) Body-centred subdivision, with 12 simplices in a BCC (body-centred cubic) lattice (Fig. 9.3(a)).

   This subdivision shares simplices between two adjacent voxels, reducing the average number of simplices per cell to 12. However, it fails criterion i): that interpolation should be restricted to the voxel itself.

d) Face-centred subdivision, with 24 simplices, constructed by subdividing the voxel into face-centred square pyramids (Fig. 9.4(a))



(a) The Subdivision (b) Artefacts

Figure 9.3: Body-centred subdivision: 24 simplices / voxel

(a) The Subdivision (b) Artefacts

Figure 9.4: Face-centred subdivision: 24 simplices / voxel

This subdivision fails criterion iii) by requiring an average of 4 interpolated data points per voxel, and condition iv) (it has the largest magnification factor of all subdivisions considered. However, as seen in Fig. 9.1(b), Fig. 9.2(b), & Fig. 9.4(b), it fulfills condition iv), which neither a) nor b) does.

I implemented each of these subdivisions for comparison purposes: the choice of which one to use depends on the problem from which the data derives. If the features of the data are large relative to the spacing of the samples, then artefacts such as those shown in Figs. 9.1(b), 9.1(c), and 9.2(b) are much less prominent. Either the minimal subdivision or the axis-aligned subdivision can be used.

If, on the other hand, features are closely spaced (as in Fig. 9.1(b) and Fig. 9.1(c)), then the face-centred subdivision (24 simplices / voxel) has fewer unpleasant artefacts than any of the other subdivisions, but requires significantly more memory and processing time to generate isosurfaces.

In Sec. 9.7, I compare the minimal subdivision scheme with Marching Cubes (Sec. 3.1).

9.2 Boundary Effects

Both van Kreveld et al. [26] and Tarasov & Vyalyi [23] assume that the contours may extend to the boundaries of the dataset: this complicates their algorithms for processing the contour tree, results in open surfaces, and adds additional splits and joins in the contour tree.

In the early stages of developing the algorithm given above, I was concerned about boundary effects. As a result, I chose to avoid them by embedding the entire dataset in a layer of zeroes (or some other value smaller than all values in the dataset). This not only avoided boundary cases, but also reduced the number of splits and joins, and guaranteed that all surfaces will be closed topologically. The extra layer of cells was rendered as well as the original dataset. However, since the interpolation inside the original data set is unaffected by the embedding process, the outer layer could have been suppressed for rendering.

Instead of automatically adding the zero layer internally, I added the zeroes to the file in which the data was stored: the zeroes could instead be added when reading in the data in $O(n)$ time.

This zero-embedding step turned out to be unnecessary: Algorithm 7.2 operates correctly at the boundary. In an implicitly-represented mesh such as I used, some special handling would still be required, since boundary vertices do not have the same connectivity as interior vertices.

9.3 Symbolic Perturbation of Data

As noted above, I assume that the data values are unique (Assn. 1.5): that no two vertices have the same value associated with them. This assumption is necessary for the guarantee that the critical points occur at the vertices (Sec. 4.1.1).

For the sake of simplicity, I chose to perturb the data symbolically, by

adding an ϵ to each value proportional to its location in RAM [11]: when comparing values, ties are broken by comparing memory addresses. This results in a stable sort order, provided we do not move data around in memory. Otherwise, this form of symbolic perturbation may give inconsistent results. Since I assume an array of data, which is initialized at the beginning of the program, and never moved around, this does not pose a problem.

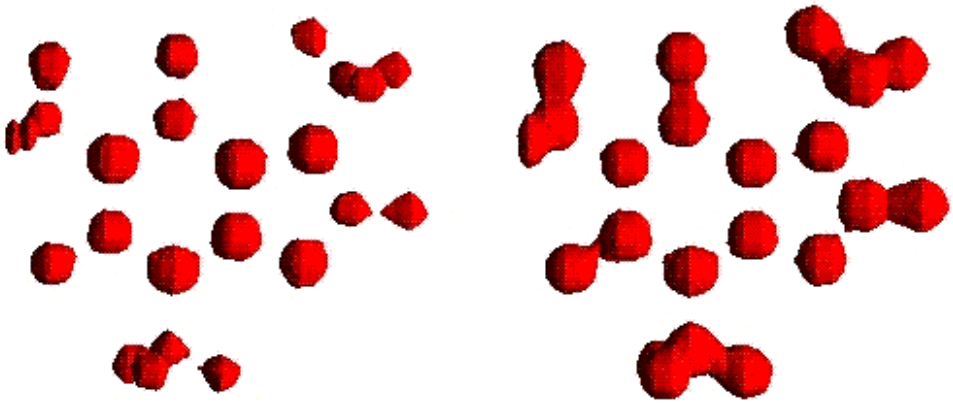
A minor complication is added in the zero-embedding stage: If the global minimum is adjacent to the zero-embedding layer, but not to the element of that layer which is adjacent to the global minimum in the sorted list, one or more spurious joins will be added in the zero-embedding layer. This was resolved by special case treatment of the zero-embedding layer, which can be assumed to belong to one component. As noted in Sec. 9.2, the zero-embedding is not required, so this complication can be avoided.

9.4 Searching for Interpolating Edge

To generate an actual level set, it is necessary to use the contour tree to find seeds to start the contour-following algorithm. This is discussed in more detail in Sec. 8.

9.5 Local Contours

Since contour trees preserve topological information about the entire dataset, it proved possible to generate contours locally around individual local maxima



(a) Local Contours (b) Fixed Contours

Figure 9.5: Comparison of Local and Fixed Contours

(see Fig. 9.5(a)): this provides better resolution of individual peaks than a single level set can do (Fig. 9.5(b)). Further exploration of this technique is desirable.

9.6 Memory Requirements

Due to some early design choices, memory overhead of the current implementation is significant. For each vertex, I store the following information:

1. *type*: whether the vertex is one of the original data points, an interpolated vertex on a face of a voxel, or an interpolated vertex in the centre of a voxel

2. *value*: the value at the vertex

3. *3 tree nodes*: pointers to the vertex in the contour tree, join tree and split tree: the data types defined for edges and vertices are themselves inefficient

4. *ID*: an ID number for the node

5. *queue flag*: flag marking whether the vertex is on the leaf queue

6. *follow flags*: flags for which simplices have been visited in the voxel based at this vertex.

In the case of the body-centred and face-centred subdivisions (Sec. 9.1.2), this is increased by the need to store the interpolated vertices, if only for sorting. This was achieved in practice by doubling each dimension of the grid, increasing storage requirements by a factor of 8. Thus, the face-centred subdivision uses approximately 800 bytes per vertex at present on a 32-byte machine.

Clearly, this is excessive: I estimate that the amount of memory required is about 10-20 bytes per vertex for the minimal subdivision, and perhaps as much as 100 bytes for the face-centred subdivision. This will be affected, however, by the complexity of the contour tree, which we saw to be $O(t)$ (in the worst case, $O(n)$)[1].

File: atom9.txt
Grid size: 13x13x13
1728 voxels; 8640 simplices.
Memory required: 219,700 bytes.
9.33 milliseconds to sort data.
12.07 milliseconds to build join and split trees.
0.23 milliseconds to merge join and split trees.
19 nodes in join tree; 4 nodes in split tree
21 nodes in contour tree.

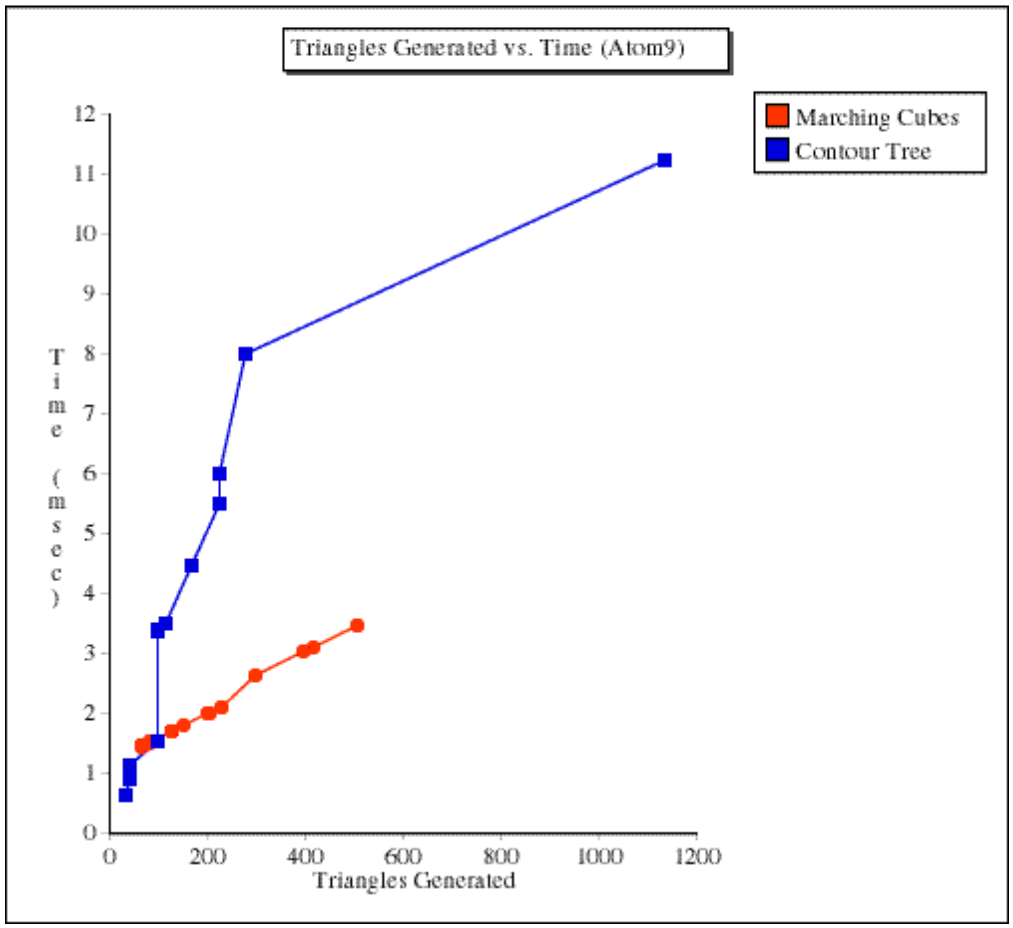
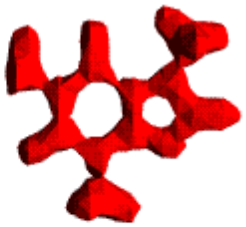


Figure 9.6: Timing Results for the Atom9 dataset

File: caffeine2.txt
Grid size: 19x19x11
3240 voxels; 16200 simplices.
Memory required: 397,100 bytes.
18.24 milliseconds to sort data.
24.42 milliseconds to build join and split trees.
0.54 milliseconds to merge join and split trees.
37 nodes in join tree; 4 nodes in split tree.
39 nodes in contour tree.



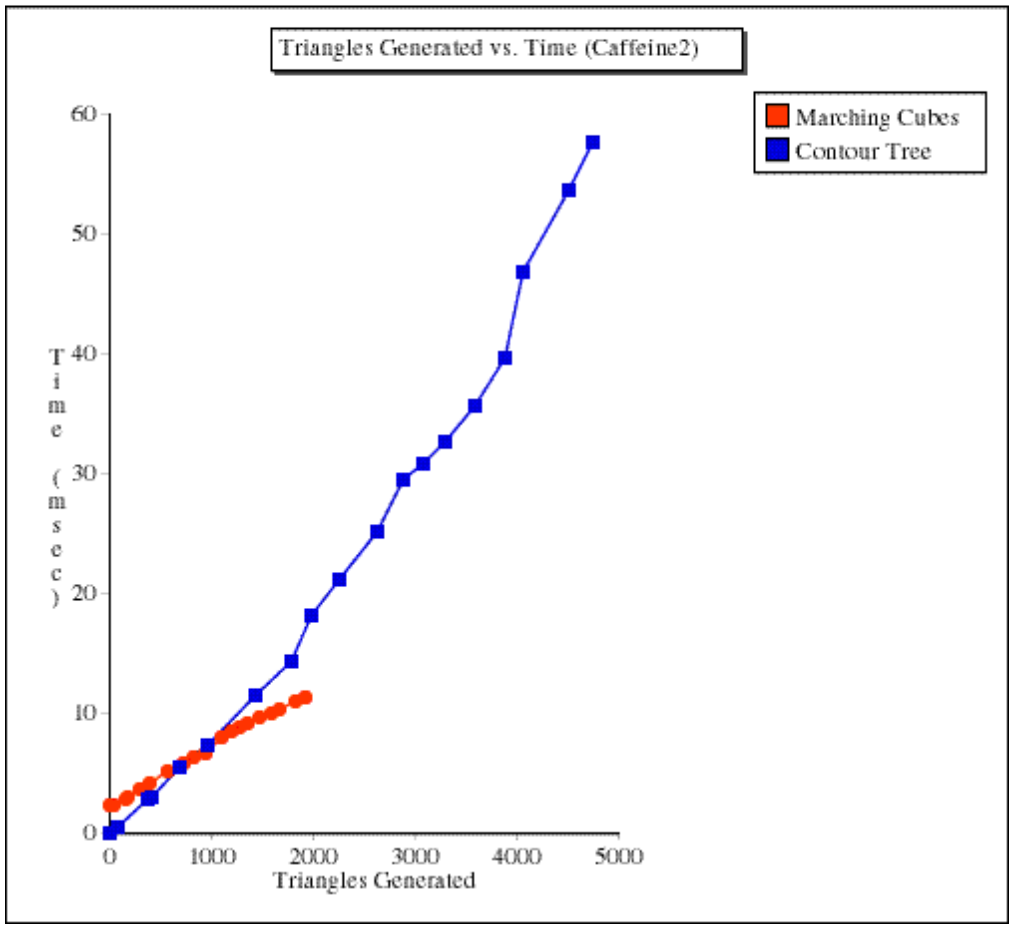
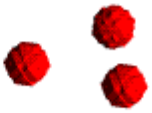


Figure 9.7: Timing Results for the Caffeine2 dataset

File: 29g.txt
Grid size: 53x53x53
140,608 voxels; 703,040 simplices.
Memory required: 14,887,700 bytes.
9.33 milliseconds to sort data.
12.07 milliseconds to build join and split trees.
0.23 milliseconds to merge join and split trees.
19 nodes in join tree; 4 nodes in split tree
21 nodes in contour tree.



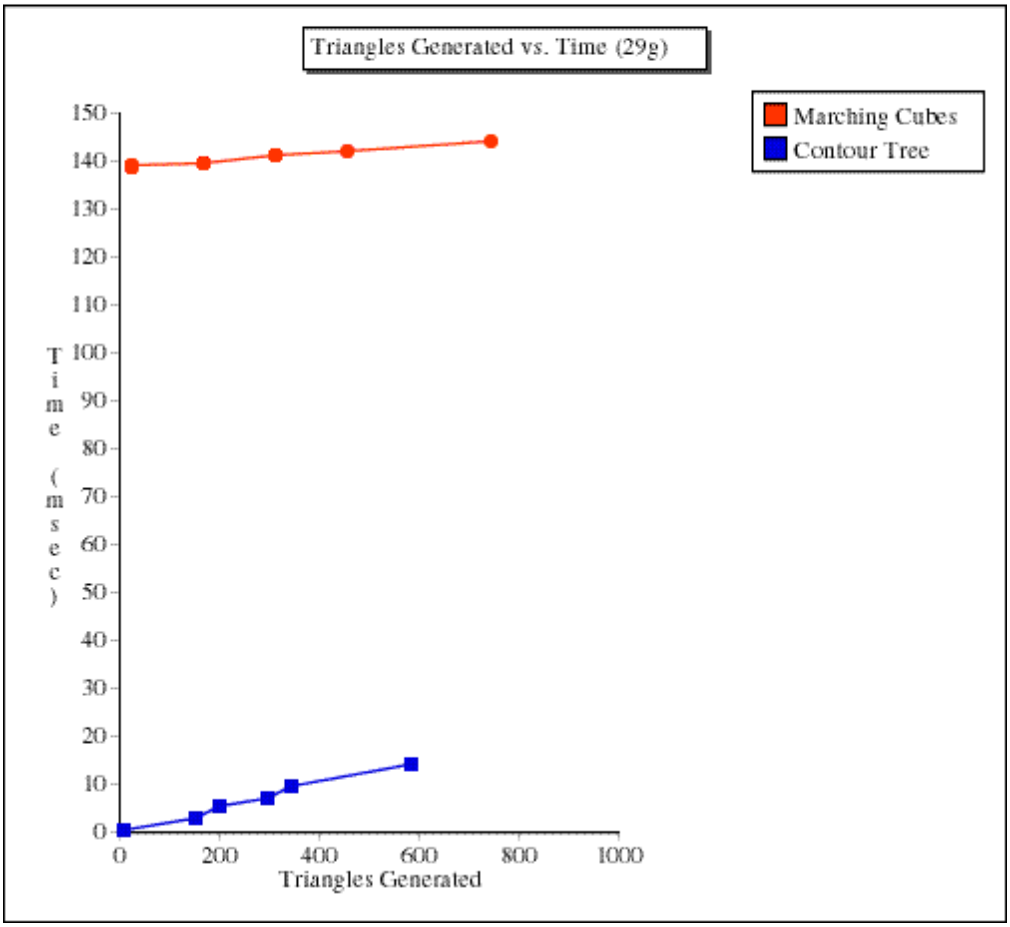


Figure 9.8: Timing Results for the "29g" dataset

9.7 Timing

In Figs. 9.6, 9.7, and 9.8, I give some sample times for the implementation. The timing was run on a 300 MHz Power Macintosh for two reasons. The current operating system (MacOS 8.1) is non-pre-emptive, so it can be assumed that no other processes interrupted the test runs: thus, timing should be quite accurate (I rarely observed a deviation of more than about 5%). Secondly, a convenient function exists to give the current clock value in microseconds since start-up, providing fine granularity of measurements.

Since the interface for the implementation used OpenGL, I conducted timing runs by measuring how long it took to construct a display list to render the isosurface, with normals. This decoupled actual rendering from the cost of moving through the mesh to locate and generate triangles.

A simple version of the marching cubes algorithm was implemented for comparison. This consisted of nested loops to march through all the cells in the mesh, generating triangles in any that intersected the isosurface.

In contrast, the contour-following algorithm takes two passes through the data: the first pass generates the triangles, marking the cells that have been visited, while the second pass unmarks the cells. Due to the memory issues referred to above (Sec. 9.6), timing was performed using the minimal subdivision.

All available optimizations were enabled, including inlining. However, the contour-following routine can be assumed not to be optimized, as it is im-

[1] see Def. 2.10, p.13 and Def. 2.7, p.12 for definition of the parameters.

plemented recursively: thus, the times shown include overhead for the function calls.

Each display list was created 100 times, then immediately destroyed: the average of the results is shown: deviation from the average was small. Also shown is the triangle count for both algorithms, the number of cells visited by the contour-following algorithm, and the maximum depth of recursive call.

Isosurfaces were generated at 5% intervals, based on the maximum value in the dataset: thus, 19 results are shown for each dataset. Note that Marching Cubes tends to be faster than Contour Following for large isosurfaces, but slower for small ones.

Also note that the two algorithms do not produce the same isosurfaces, since the interpolation inside each voxel differs. In particular, Marching Cubes generates fewer triangles in most cases. This is not necessarily an advantage, since contour-following can potentially generate long triangle strips (see Sec. 3.1.3, p.16 and Sec. 3.5.2, p.22), reducing the cost of sending the triangles to hardware. Although triangle strips were not implemented, the depth of the recursive call tended to be approximately 2/3 of the total number of cells visited (see Figs. 9.6, 9.7, and 9.8). This indicates that very long triangle strips are possible in practice.

Finally, a word about the data sets: since the implementation was on a small scale, and real crystallographic datasets are generally much larger, a simple simulation was used: a helper program created data sets from a list of atomic nuclei and their positions. The caffeine dataset (Fig. 9.7) was

created by taking a standard crystallographic PDB datafile, and can be seen in Fig. 9.5. The 9 atom dataset (Fig. 9.6) was created by placing 9 nuclei of different radii in entirely arbitrary positions. It was particularly useful for showing artefacts in the data, and can be seen in Fig. 9.4(b). The last dataset (Fig. 9.8), the largest, consisted of three isolated nuclei. Thus, the isosurfaces are smaller than would normally be the case: this dataset was included to give some idea of the potential speed advantage over the marching cubes algorithm for datasets where $N >> k$.

Although these results are far from rigorous, it is clear that the contour-following algorithm is worth implementing for large datasets.

Chapter 10

Summary

3-D datasets naturally arise in various application fields. Some fields, such as X-ray crystallography, require interactive visualization of large datasets.

One technique to visualize such datasets is the use of contour trees to construct isosurfaces. This thesis describes a practical and efficient algorithm for constructing contour trees for 3-Ddatasets. Details of an implementation were sketched, and the technique compared with other techniques.

In particular, if we review the properties listed in Sec. 1.3, p.6, we see that contour tree techniques satisfy all of the properties desired for dealing with X-ray crystallographic data (see Sec. 9.5, p.98 for details on the last item).

Chapter 11

Extensions

A number of extensions are possible in practice. These include: dimensions other than 3-D, application to irregular meshes, flexible contours, the use of transparent shells for visualization, and scaling issues.

11.1 Higher and Lower Dimensions

Although the discussion throughout has focussed on 3-D datasets such as those acquired in X-ray crystallography, the algorithm depends in no way on this assumption. Implementation details will vary slightly in higher or lower dimensions, but the construction outlined above remains valid: the algorithm is equally useful in 2, 3, 4, or more dimensions, although due to the size of datasets, 4-Dmay be the practical limit.

Most of the other algorithms described in Ch. 3 can be modified to work in higher dimensions. Marching Cubes (Sec. 3.1, p.14) would become

Marching Hypercubes, and would have 65536 cases before symmetry. Most of the other algorithms would scale rather more easily.

11.2 Irregular Meshes

This thesis assumed that the data was presented on a regular mesh (Assn. 1.2). As with dimensionality (Sec. 11.1), this assumption is unnecessary: the algorithm as outlined works equally well for irregular meshes. I expect to implement a version for irregular meshes at some future date.

Again, most of the algorithms described in Ch. 3 can be modified to work on irregular meshes, with more or less success.

11.3 Flexible Contours

Local contours (Sec. 9.5) have been implemented to show the immediate neighbourhood at local maxima. So far, this has only been implemented for the superarc incident to each local maximum. To do this, an isovalue is interpolated along the superarc incident to the local maximum, and the corresponding contour generated: this fulfills the goal expressed in Sec. 1.3, p.6 for crystallography. This needs further development, particularly with respect to user-interface issues, such as when and how to merge contours as the isovalue passes a supernode.

Note that Marching Cubes, and all of the other algorithms in Ch. 3 except for the contour-following algorithms (Sec. 3.5, p.20) are unable to gen-

erate flexible or local contours, without significant additional processing time.

11.4 Transparent Shells

A simple extension to the use of isosurfaces is the use of multiple transparent isosurfaces or shells. As with contour lines on a map, this permits the entire dataset to viewed simultaneously, rather than a single slice. This idea has surfaced in the literature (e.g. in Guo [13]), but is not well-developed.

11.5 Scaling And Parallelism

Datasets in 3-D can contain as many as 1000^3 data points, and may not fit into available memory. In the future, I will be researching ways to scale the algorithm to perform efficiently on such large datasets. One way that seems promising is to use parallel processors to compute partial join trees or contour trees, then merge the results.

Bibliography

[1] Chandrajit L. Bajaj and Valerio Pascucci. Progressive IsoContouring. Technical Report 99-36, Texas Institute of Computational and Applied Mathematics, Austin, Texas, 1999.

[2] Chandrajit L. Bajaj, Valerio Pascucci, and Daniel R. Schikore. Seed Sets and Search Structures for Optimal Isocontour Extraction. Technical Report 99-35, Texas Institute of Computational and Applied Mathematics, Austin, Texas, 1999.

[3] Thomas F. Banchoff. Critical Points and Curvature for Embedded Polyhedra. *Journal of Differential Geometry*, 1:245–256, 1967.

[4] Jon Louis Bentley. Multidimensional Binary Search Trees Used for Associative Searching. *Communications of the ACM*, 18(9):509–517, 1975.

[5] Paolo Cignoni, Paola Marino, Claudio Montani, Enrico Puppo, and Roberto Scopigno. Speeding Up Isosurface Extraction Using Interval Trees. *IEEE Transactions on Visualization and Computer Graphics*, 3(2):158–169, 1997.

[6] Paolo Cignoni, Claudio Montani, Enrico Puppo, and Roberto Scopigno. Multiresolution Representation and Visualization of Volume Data. *IEEE Transactions on Visualization and Computer Graphics*, 3(4):352–369, 1997.

[7] Kevin Cowtan. Book of Fourier. *http://www.yorvic.york.ac.uk/~cowtan/fourier/fourier.html*, 1994.

[8] Mark de Berg and Marc van Kreveld. Trekking in the Alps Without Freezing or Getting Tired. In *First Annual European Symposium on Algorithms (ESA '93)*, pages 121–132, 1993.

[9] Jan Drenth. *Principles of Protein X-ray Crystallography*. Springer-Verlag, Berlin; Heidelberg; New York, 1994.

[10] Herbert Edelsbrunner. Dynamic Data Structures for Orthogonal Intersection Queries. Technical report, Inst. Informationsverarb, Tech. Uniz. Graz, Graz, Austria, 1980.

[11] Herbert Edelsbrunner and E.P. Mücke. Simulation of Simplicity: A technique to cope with degenerate cases in geometric algorithms. *ACM Transactions on Graphics*, 9(1):66–104, 1990.

[12] H. Freeman and S.P. Morse. On Searching A Contour Map for a Given Terrain Elevation Profile. *Journal of the Franklin Institute*, 284(1):1–25, 1967.

[13] Baining Guo. Interval Set: A Volume Rendering Technique Generalizing Isosurface Extraction. In *IEEE Proceedings on Visualization 95*, pages 3–10. IEEE, 1995.

[14] C.T. Howie and Edwin H. Blake. The Mesh Propagation Algorithm for Isosurface Construction. *Computer Graphics Forum*, 13:65–74, 1994.

[15] Takayuki Itoh and Koji Koyamada. Automatic Isosurface Propagation Using an Extrema Graph and Sorted Boundary Cell Lists. *IEEE Transactions on Visualization and Computer Graphics*, 1(4):319–327, 1995.

[16] Takayuki Itoh and Koji Koyamada. Isosurface Extraction By Using Extrema Graphs. *IEEE Transactions on Visualization and Computer Graphics*, 1:77–83, 1995.

[17] Yarden Livnat, Han-Wei Shen, and Christopher R. Johnson. A Near Optimal Isosurface Extraction Algorithm Using the Span Space. *IEEE Transactions on Visualization and Computer Graphics*, 2(1):73–84, 1996.

[18] William E. Lorenson and Harvey E. Cline. Marching Cubes: A High Resolution 3D Surface Construction Algorithm. *Computer Graphics*, 21(4):163–169, 1987.

[19] Gregory M. Nielson and Bernd Hamann. The Asymptotic Decider: Resolving the Ambiguity in Marching Cubes. In *IEEE Proceedings on Visualization 91*, pages 83–91. IEEE, 1991.

[20] Paul Ning and Jules Bloomenthal. An Evaluation of Implicit Surface Tilers. *IEEE Computer Graphics and Applications*, 13:33–41, 1993.

[21] Max Perutz. *Protein Structure*. W.H. Freeman & Company, New York, 1992.

[22] Han-Wei Shen and Christopher R. Johnson. Sweeping Simplices: A fast iso-surface extraction algorithm for unstructured grids. In *IEEE Proceedings on Visualization 95*, pages 143–150. IEEE, 1995.

[23] Sergey P. Tarasov and Michael N. Vyalyi. Construction of Contour Trees in 3D in $O(n \log n)$ steps. In *Proceedings of the 14th ACM Symposium on Computational Geometry*, pages 68–75. ACM, 1998.

[24] R.E. Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22:215–225, 1975.

[25] Marc van Kreveld. *Digital Elevation Models and TIN Algorithms*. In *Algorithmic Foundations of Geographic Information Systems*, pages 37–78 Springer-Verlag, Berlin; Heidelberg; New York, 1997.

[26] Marc van Kreveld, René van Oostrum, Chandrajit L. Bajaj, Valerio Pascucci, and Daniel R. Schikore. Contour Trees and Small Seed Sets for Isosurface Traversal. In *Proceedings of the 13th ACM Symposium on Computational Geometry*, pages 212–220. ACM, 1997.

[27] Jane Wilhelms and Allen van Gelder. Octrees for Faster Isosurface Generation. *ACM Transactions on Graphics*, 11(3):201–227, 1992.

[28] Geoff Wyvill, Craig McPheeters, and Brian Wyvill. Data Structure for Soft Objects. *Visual Computer*, 2:227–234, 1986.