1997

Seed Sets and Search Structures for Accelerated Isocontouring

Chandrajit L. Bajaj

Valerio Pasucci

Daniel R. Schikore

Report Number:
97-034

SEED SETS AND SEARCH STRUCTURES FOR ISOCONTOURING

Chandrajit L. Bajaj
Valerio Pascucci
Daniel R. Schikore

Department of Computer Sciences
Purdue University
West Lafayette, IN 47907

CSD-TR #97-034
June 1997

Seed Sets and Search Structures for Isocontouring

Chandrajit L. Bajaj[†] Valerio Pascucci[†] Daniel R. Schikore[†]

Shastra Lab & Center for Image Analysis and Data Visualization
Department of Computer Sciences
Purdue University

Abstract

We present three algorithms for the construction of *seed sets*, a subset of a cell complex which contains at least one cell for each connected component of each isocontour, for all possible isovalues. Seed sets reduce the storage requirements of high performance search structures for isocontouring, such as the segment tree or the interval tree. The three algorithms determine seed sets with varying properties. The first computes seed sets within a constant factor of the optimal size, requiring $O(n \log n)$ time, where n is the size of the mesh. A more conservative approach computes seed sets of slightly larger size with $O(n)$ processing time, and is very amenable to parallel processing. The seeds produced follow a particular pattern which can be leveraged for performing out-of-core isocontouring, dynamically loading only the data which is necessary from secondary storage or a remote server. A specialized form of the second algorithm for regular grids computes seed sets of intermediate size, with only slightly additional effort and the same computational complexity. We examine the use of three search structures and compare their application to the seed sets and full sets of cells from a variety of computational grids.

1 Introduction

Isocontouring is a widely used approach to the visualization of scalar data and an integral component of almost every visualization environment. Computation of isocontours has applications in visualization ranging from extraction of surfaces from medical volume data [16] to computation of stream surfaces for flow visualization [32]. Inherent in the selection of an isocontour, defined by $C(w) : \{\mathbf{x} | \mathcal{F}(\mathbf{x}) - w = 0\}$, is that only a selected subset of the data is represented in the result. In many applications, the ability to interactively modify the isovalue w while viewing the computed result is of great value in exploring the global scalar field structure. In fact, it has been observed in user studies that the majority of the time spent interacting with a visualization is in modifying the visualization parameters, *not* in changing the viewing parameters [8]. Hence there has been great interest in improving the computational efficiency of contouring algorithms.

In the following subsections, we divide the principal components of cell-based isocontouring algorithms in the following three stages:

- Cell Triangulation – Method of computation for determining the component of a contour which intersects a single cell.
- Cell Search – Method for finding all cells which contain components of the contour
- Cell Traversal – Order of cell visitation may be integrated with (or decided by) the cell search technique, however it nevertheless affects the performance of the isocontour extraction algorithm

In Section 1.1 we discuss the problem of cell triangulation and review several approaches which have been presented in the literature. In Sections 1.2 and 1.3 we present the prior work on search and traversal schemes which provides motivation for our work. We demonstrate that combinations of the approaches in each of these three areas yield dramatic improvements in the interactivity of isocontouring with a small overhead in the required data structures.

1.1 Cell Triangulation

Cell triangulation concerns the approximation of the component of a contour which is interior to the cell. Triangulation has two distinct components, *interpolation* to determine a set of points and normals, and *connectivity* to determine the local topology of the contour. While the use of linear interpolation edges of cells is a widely accepted approach, other strategies have been developed to reduce this computational portion of isocontour approximation, such as selecting midpoints along intersected edges [21]. Here we present a summary of the work on the topological aspect of contour triangulation.

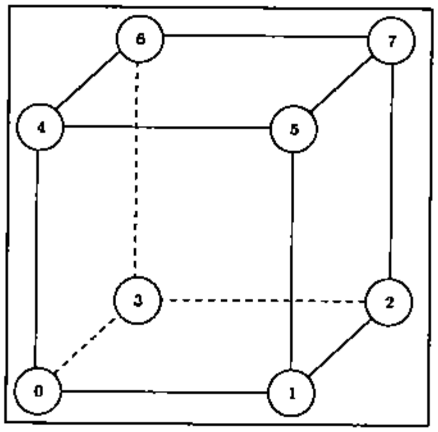


Figure 1: Standard cell representation for contour computation in a structured grid

Cell-based contouring algorithms generally begin with a classification of each vertex of a given cell as *positive* (if greater than the isovalue) or *negative* (if less than or equal to the isovalue), which we will refer to as *black* and *white*, respectively. Such a binary classification of the 8 vertices of a regular cell (as in Figure 1) leads to a total of 2^8 or 256 possible configurations. Taking rotational symmetry into account, this can been reduced to 22 distinct cases [15, 27]. Based on linear interpolation, cell edges are called *intersecting* if the colors of the endpoints differ, or *non-intersecting* if they are colored the same.

Marching Cubes [17] further reduces the number of base cases by assigning complementary triangulation for complementary vertex configurations (*black* to *white*), resulting in 15 distinct colorings, as shown in Figure 2. The full table of the 256 possible vertex configurations can easily be generated from this table of 15 cases.

The use of complementary triangulations reduces the number of base cases, but also introduces a well-know topological inconsistency on certain configurations of shared faces between cubes [6], one case of which is illustrated in Figure 3. A number of techniques have been proposed which offer solutions to this inconsistency, which we group into two classes. The first class attempts only to provide *consistency* along all cell faces, while then second class provides *correctness* with respect to a chosen model.

Consistency may be achieved simply by subdividing each cell into tetrahedra and using a linear interpolant within each tetrahedron [5]. An efficient approach to consistency is to adopt a consistent decision rule, such as sampling the function at the center of the ambiguous face to determine the local topology [36].

Zhou et al. make the point that the tetrahedral decomposition and linear approximation change the function and may still result in incorrect, though consistent, topology [37]. They propose that the tetrahedral decomposition may be used, provided that intersections along the introduced diagonal are computed for the cubic function which results from sampling the trilinear function across the diagonal of a cell, rather than applying linear interpolation along all edges.

Matveyev sorts all intersections on a regular cell face with respect to an axial direction [18]. The nature of the bilinear interpolant ensures that pairs in the sorting will be connected, as the asymptotes at a saddle point for a bilinear function over an axis-aligned regular cell are parallel to the axes.

The core of the problem along shared cell faces lies in determining the topological connectivity of vertices which are colored the same but which lie diagonally across a face of a cell. Nielson and Hamann propose generating a consistent decision on connectivity by enforcing a topology which is correct with respect to the bilinear interpolant along the face [24]. Kenwright derives a similar condition for disambiguating the connectivity on the faces in terms of the gradient of the bilinear interpolant [13]. Natarajan further enforces consistency with the trilinear interpolant for the case of ambiguities which are interior to a cell, which occur when diagonal vertices across the body of the cell are similarly colored but have no edge-connected path of vertices of the same color between them [23]

Karron et al. further discuss the proper treatment of criticalities in isocontouring, proposing a digital morse theory for describing scalar fields [12].

Wilhelms and Van Gelder provide a comprehensive review the topological considerations in extracting isosurfaces, and demonstrate that gradient heuristics applied at the vertices of a cell are necessary and sufficient to disambiguate the topology of functions which are quadratic [34, 28, 29].

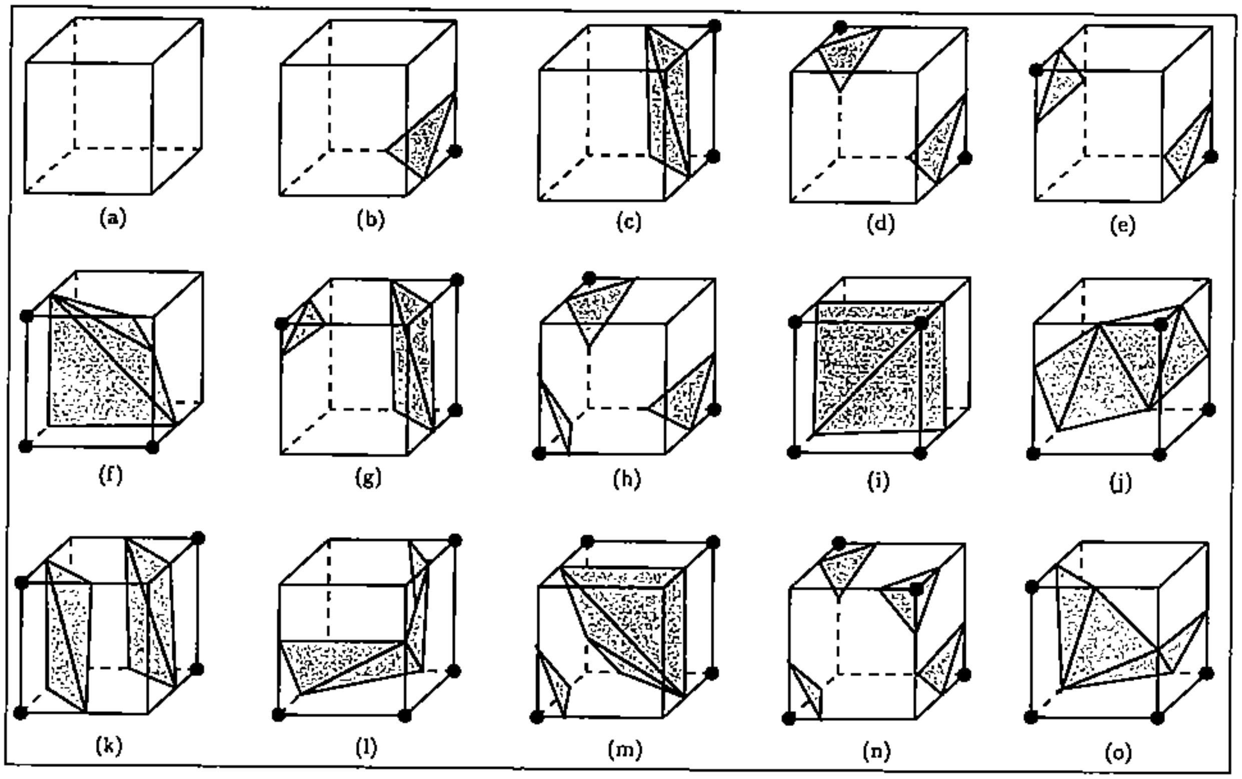


Figure 2: 15 distinct vertex colorings

The solution suggested by Natarajan [23] is particularly attractive due to its design to enforce consistency with the trilinear interpolant, a commonly used interpolant for 3d reconstruction and visualization. The situation on faces with colored vertices which are diagonally adjacent can be viewed in two dimensions as in Figure 4. The unique saddle point at coordinate $\mathbf{x}_s$ of the bilinear interpolant lies interior to the face, and the correct topology can be determined by evaluating the function at the saddle point and comparing it with the isovalue as shown. This topological consistency is carried out further by considering the unique saddle point of the full trilinear interpolant in addition to the six possible face saddles. A simple extension to the marching cubes case table requires sub-cases only for configurations which contain saddles. The sub-cases are indexed by the saddle point evaluations in order to determine a triangulation which is topologically consistent with the trilinear interpolant [23]

For the inconsistent case illustrated in Figure 3, several distinct topological triangulations are possible, two of which are illustrated in Figure 5.

1.2 Cell Search

Because a contour only passes through a fraction of the cells of a mesh on average, algorithms which perform an exhaustive covering of cells are found to be inefficient, spending a large portion of time traversing cells which do not contribute to the contour.

The straightforward approach of enumerating all cells to extract a contour leads to a high overhead cost when the surface being sought intersects only a small number of the cells.

Preprocessing of the scalar field permits the construction of search structures which accelerate the repeated action of isocontouring, allowing for increased interactivity during modification of the isovalue. Many preprocessing approaches and search structures have been presented, which are conveniently classified (similar to the classification presented in [14]) based on whether the search is in *domain* space or *range* space.

1.2.1 Domain Search

- **Octree search** – A spatial hierarchy for accelerating the search process is a natural approach which has been explored by Wilhelms and Van Gelder [35, 33]. For space efficiency considerations, a partial octree decomposition was developed which groups all cells at the highest level and adaptively approximates the

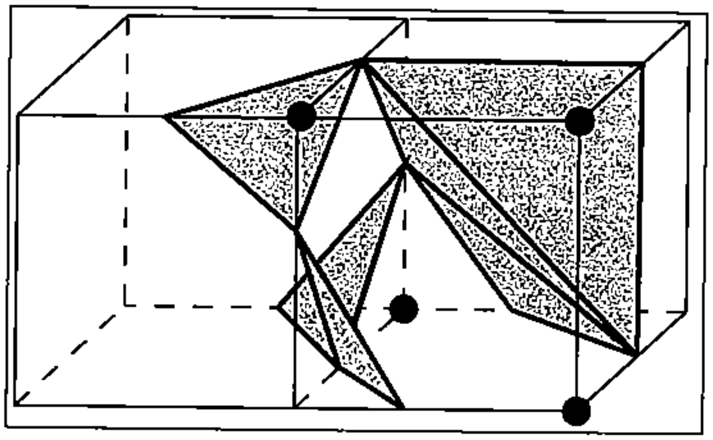


Figure 3: Topological inconsistency associated with the original marching cubes

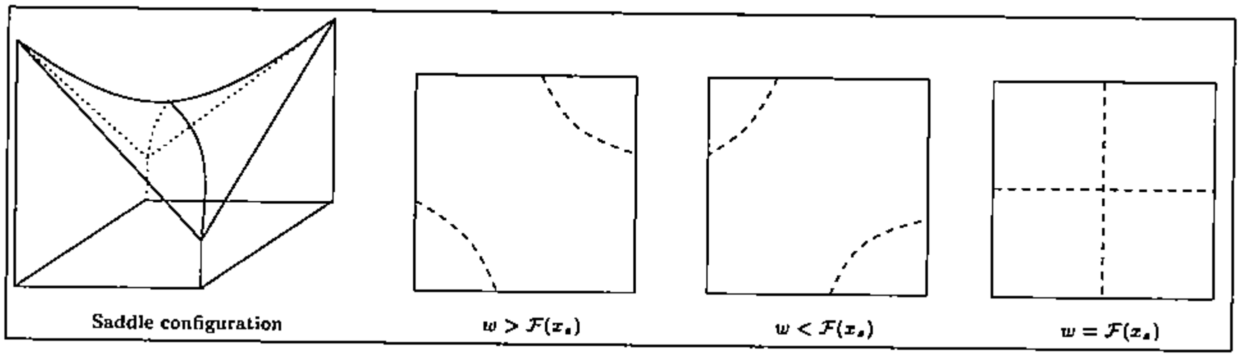


Figure 4: A two dimensional bilinear saddle and its contour configurations

data through axis-aligned subdivisions which better approximate the data. At each level in the tree, *min* and *max* values for the cells contained in the subtree are stored, providing a means to efficiently discard large spatial regions in the search phase. An analysis presented in [14] suggests a worst-case computational complexity of $O(k + k \log \frac{n_c}{k})$, where k is the size of the output and n_c is the number of cells.

1.2.2 Range Search

A large number of search techniques in the recent literature perform the search for intersected cells in the *range* space of the function. As we are dealing only with scalar-valued functions, range space search techniques have the advantage of being independent of the dimension of the domain. In range space, each cell c is associated with the continuous set of values taken on by the function over the domain:

$$R(c) = [\min_{x \in c} \mathcal{F}(x), \max_{x \in c} \mathcal{F}(x)]$$

There are two approaches for representing the range space, the 1D *value*-space, in which each range $R(c)$ is considered as a segment or interval along the real line, and the 2D *span-space*, in which each range $R(c)$ is considered as a point in 2D [14], as illustrated in Figure 7. While certain search structures are motivated by one geometric representation or another, others may be effectively visualized in either representation.

We present a brief summary of the range space approaches which have been proposed in the literature.

- **Min-Max lists** – Giles and Haimes introduce the use of *min-max* sorted lists of cells to accelerate searching. In addition to forming two sorted lists of cells, the maximum cell range, Δw, is determined. Cells containing an isosurface of value w must have minimum value in the range $[w - \Delta w, w]$, which may be determined by binary search in the *min*-sorted array. This *active set* of cells is purged of cells whose range does not contain w. For small changes in w, the active list can be updated, rather than wholly recomputed, by adding and purging new candidate cells to the active list. In the worst case, complexity remains $O(n_c)$.

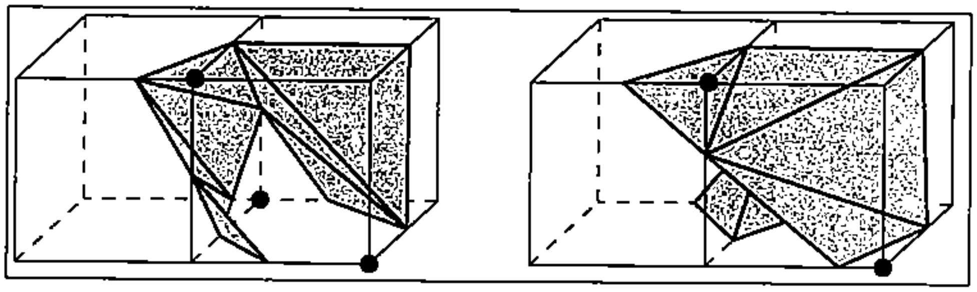


Figure 5: Two topologically consistent triangulations with respect to the shared face. Note that additional distinct topological configurations exist due to additional face saddles on the non-shared faces

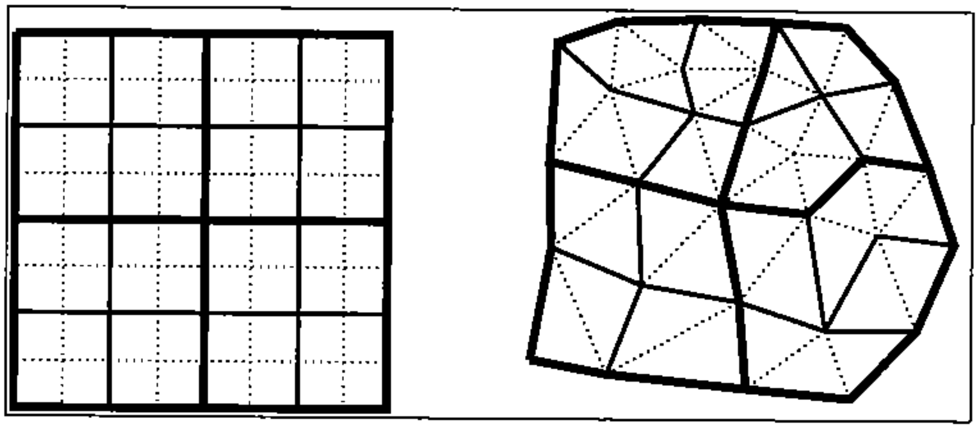


Figure 6: Spatial hierarchical cell decompositions for accelerating the search for isocontours.

- **Span filtering** – Gallagher describes an algorithm called span filtering[7], in which the entire range space of the scalar function is divided into a fixed number of *buckets*. Cells are grouped into buckets based on the minimum value taken on by the function over the cell. Within each bucket, cells are classified into one of several lists, based on the number of buckets which are *spanned* by the range of the cell. For an individual isovalue, cells which fall into a given bucket need only be examined if their span extents to the bucket which contains the isovalue. In the worst case, complexity remains $O(n_c)$.
- **Sweeping simplices** – Shen and Johnson describe a *Sweeping Simplices* algorithm [26], which builds on the *min-max* lists of Giles and Haimes and augments the approach with a hierarchical decomposition of the value-space. The *min*-sorted list is augmented by pointers to the associated cell in the *max*-sorted list, and the *max*-sorted list is augmented by a "dirty bit." For a given isovalue, a binary search in the *min*-sorted list determines all cells with minimum value below the isovalue. Pointers from the minimum value list to the maximum value list are followed to set the corresponding dirty bit for each candidate cell. At the same time, the candidate cell with the largest maximum value which is less than the isovalue is determined. As a result, all marked (candidate) cells to the right of this cell in the maximum list must intersect the contour, as they have minimum value below the isovalue and maximum value above the isovalue. Optimizations may be performed when the isovalue is changed by a small delta. One *min-max* list is created for each level of a hierarchical decomposition of the *min-max* search space. The overall complexity remains $O(n_c)$ in the worst case analysis.
- **Extrema graphs** – Itoh and Koyamada compute a graph of the extrema values in the scalar field [10]. Every connected component of an isocontour is guaranteed to intersect at least one arc in the graph. Isocontours are generated by propagating contours from a seed point detected along these arcs. Noisy data with many extrema will reduce the performance of such a strategy. Livnat et al. note that in the worst case the number of arcs will be $O(n_c)$, and hence straightforward enumeration of the arcs is equivalent in complexity to enumeration of the cells.
- **Kd-tree** – Livnat, Shen, and Johnson describe a new approach which operates on the 2D *min-max span space* [14]. Cells are preprocessed into a *Kd-tree* which allows $O(k+\sqrt{n_c})$ worst case query time to determine

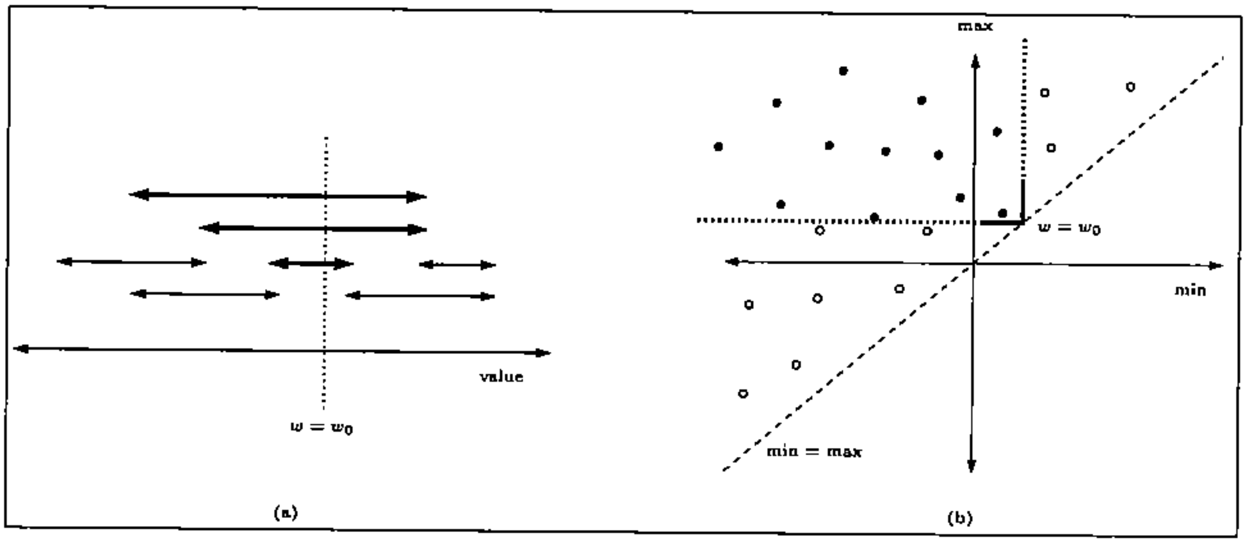


Figure 7: The (a) 1D value space and (b) 2D span space representations for range-space searches

the cells which intersect the contour, where k is the size of the output. It is reported that in the average case, k is the dominant factor, providing optimal average complexity.

- **Lattice search** – The same authors, with Hansen, have described a technique which demonstrates improved empirical results by using an $L \times L$ lattice search decomposition in span space, in addition to allowing for parallel implementation on a distributed memory architecture [25]. With certain assumptions on the distributions of points in the span space, the worst-case query time improves to $O(k + \frac{n_c}{L} + \frac{\sqrt{n_c}}{L})$.
- **Segment tree, interval tree** – Several authors have recently developed improved worst-case performance bounds with the use of the *interval tree* and *segment tree* data structures. Both structures provide a search complexity of $O(k + \log n_u)$, where n_u is the number of unique extreme values of the segments which define the tree and k is the number of reported segments intersected. In Bajaj et al. a segment tree is constructed for a reduced set of *seed cells* which are extracted in a preprocessing stage [2]. van Kreveld also developed seed sets for the specialized case of a triangular mesh in two dimensions [30]. The *interval tree* used by van Kreveld provides the same search complexity with lower worst-case storage overhead, which we will examine in Section 3. Cignoni et al. use an interval tree constructed for the entire set of cells in a tetrahedral complex [3]. More recently Cignoni et al. extend their approach to efficiently handle large regular grids by building an interval tree for a specialized subset of the cells [4].

1.3 Cell Traversal

The order in which cells are visited can impact the efficiency of contouring algorithms in several ways. In the algorithms described above, cells may be traversed in marching order, through contour propagation (breadth first in a connected component), or in random order. One issue is the efficiency of avoiding re-computation (recomputing intersection along shared edges of cells). Through marching order and contour propagation, information can be saved more efficiently than in a random order visitation which is caused by some search techniques.

Contour propagation [1, 9, 10, 2] is a surface tracking method which is based on continuity of the scalar field, and hence of the isocontours derived from the field. Given a single *seed cell* on a connected component of a contour, the entire component is traced by breadth-first traversal through the face-adjacencies. The traversal is terminated when a cell which has already been processed is met again, which is usually determined by a set of *mark* bits, which indicate for each cell whether processing has taken place. The procedure is illustrated in Figure 8. In a contour propagation framework, as in a marching order traversal, optimization can be performed based on the fact that with each step, information from adjacent cells is available which can be used to avoid recomputation. In addition, the extracted contours are more easily transformed into representations such as triangle strips for efficient storage and rendering.

Several of the cell search techniques presented above depend upon a subsequent cell traversal algorithm such as contour propagation. The use of a subsequent cell traversal algorithm allows a reduction in the size of the

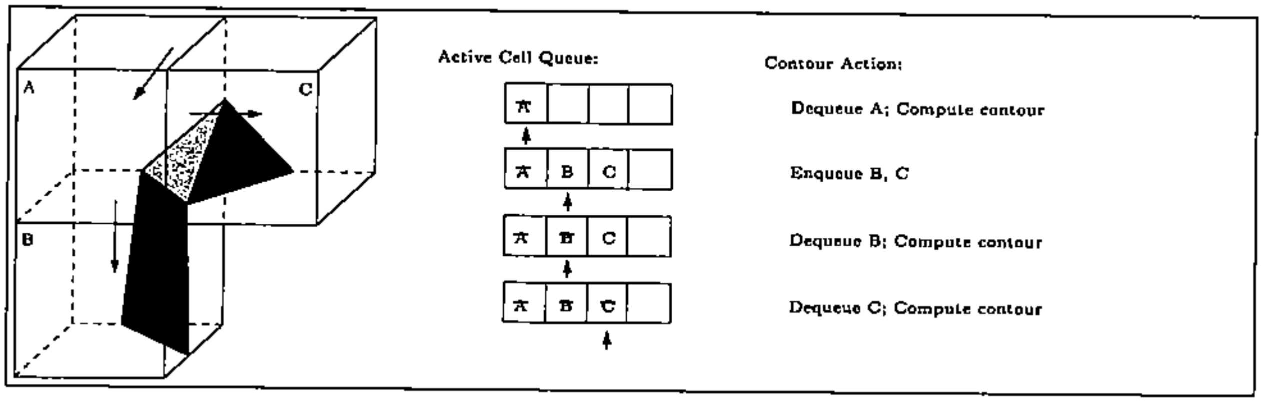

A
B
C
Active Cell Queue:
Contour Action:
Dequeue A; Compute contour
Enqueue B, C
Dequeue B; Compute contour
Dequeue C; Compute contour

Figure 8: Illustration of contour propagation. The active surface is traced through adjacent cells.

search structure, because a cell which will be processed by *traversal* need not be entered into the primary search structure. The traversal stage can be considered a secondary search phase.

Itoh and Koyamada compute extrema graphs of the scalar field which, combined with a search of boundary cells, guarantees that each contour component will intersect at least one seed cell [10]. More recently the same authors describe a volume thinning approach to computing a seed set, which reportedly results in smaller seed sets [11].

In [31], the theory of optimal seed sets is discussed, which suggests that optimal (minimal) seed sets can be constructed in time which is polynomial in the number of cells, though the cost for minimal seed sets remains prohibitive for most cases.

Cignoni et al. introduce a limited propagation scheme for regular grids based on a "checkerboard" seed set, as illustrated in Figure 9. By selecting a regular pattern of cells, it is guaranteed that all contours will intersect a black or grey cell. Modified contour propagation rules are applied to reach white cells from the selected black or grey cells. Determining the seed set requires very little computation, thus preprocessing is essentially limited to building the range search structure, in this case an interval tree. In Section 2.6 we will contrast this approach with the seed selection algorithms presented here.

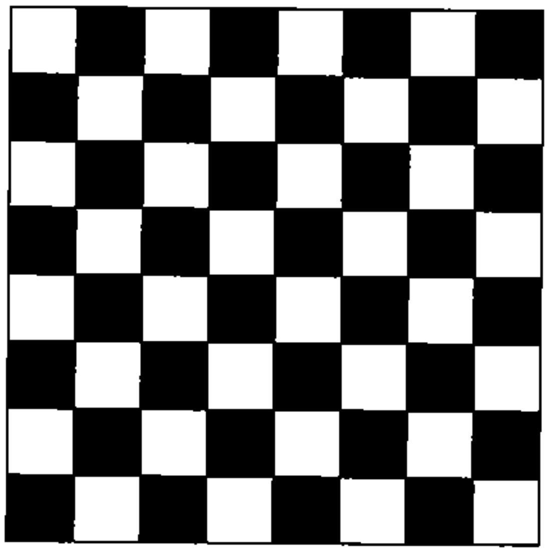


Figure 9: Illustration of the "checkerboard" approach to sufficient seed sampling. Black cells are on the checkerboard, while a number of grey cells are also required in the seed set.

1.4 Summary of Prior Work

A key to efficient computation is in exploiting coherence. The isocontouring approaches described above can be loosely classified and analyzed based on the coherence which is exploited.

Spatial Coherence – We assume a minimum of C^0 continuity in our scalar field. Continuity along shared cell faces is exploited by many contouring approaches described above. The octree decomposition exploits spatial coherence in a hierarchical manner. As should be expected, the analysis in [14] reveals that the complexity gain breaks down when the spatial frequency is high, forcing large portions of the octree to be traversed.

Range-Space Coherence – Searches in range-space have demonstrated improved worst-case query complexity with performance which is independent of spatial frequency. Such advances, however, come at the cost of decreased ability to exploit spatial coherence. Assuming a continuous scalar field over a cell representation, cells which are spatially adjacent also overlap in the value space for the range of the shared face. However, the construction of value-space search structures such as the interval tree and segment tree are completely independent of assumptions such as scalar field continuity. While this may be an advantage in the case that discontinuous fields or disjoint groups of cells are considered, for our purposes it usually means that spatial coherence is under-utilized.

We see above that spatial and value-space searches exploit coherence in one sense by sacrificing coherence in another. Our approach is best understood as a hybrid of spatial and value-space approaches, with the goal and result of exploiting both value-space and spatial coherence.

Our approach is based on a fragmentation of the search for intersected cells into *range-space* and *geometric* phases, taking advantage of coherence in both. Range-space searches exhibit improved worst-case complexity bounds due primarily to the fact that intersection of a contour with a cell is determined by range-space properties, as opposed to geometric properties. The output, however, is geometric in nature. By adopting *contour propagation* to compute each connected component, we take advantage of spatial coherence during cell traversal. Contour propagation also has the advantage of requiring only one *seed cell* for each connected component from which to begin tracing the contour. In our approach, preprocessing determines a subset S of the cells which are maintained as candidate seed cells. For an arbitrary input isovalue, it is guaranteed that every connected component of the isocontour will intersect at least one cell in S. A second preprocessing step constructs a range query structure for the cells in S. In the contour extraction phase, the contour propagation algorithm sweeps out the contour from each selected cell in the seed set. Thus, the search for intersected cells takes advantage of spatial coherence in the use of contour propagation, and range-space coherence through the construction of a range-space search structure for seed cells.

In Section 2 we present three approaches to the construction of seed sets. In Section 3 we describe three alternative data structures for performing fast queries for intersected seed cells. Together the presented algorithms and data structures provide an array of possible combinations which vary in usefulness based on the relative importance of computational, space, and query complexity.

2 Seed Set Construction

We introduce three alternatives for the construction of seed sets. Our primary concern is efficient approximation algorithms for computing "good" seed sets. The problem of optimal seed sets is considered and shown to have polynomial time complexity in [31], however the complexity may be considered excessive for many applications. As with many approximation algorithms, we find that the performance in terms of the size of the seed set can be balanced with the competing desire for low time/space complexity, resulting in three approaches which are useful in a variety of settings and applications.

In sections 2.1 and 2.2 we provide some preliminary definitions which are derived from contour propagation and give a more formal definition of seed sets, providing a foundation for seed set generation.

2.1 Cell Connectivity

We begin by extending the definitions of cell connectivity based on adjacency to encompass connectivity with respect to a given scalar value of a function defined over the domain. In this way we can identify the cells which can be reached in the cell traversal stage (i.e. by contour propagation) from those which must be part of the seed set.

Based on propagation of contours through cell adjacencies (as presented in Section 1.3), the connectivity is simply described by a labeled adjacency graph of the mesh cells. The use of a different propagation scheme would require the construction of a connectivity graph different from the adjacency graph. In general, to define the connectivity graph we assume:

1. The function $\mathcal{F}(\mathbf{x})$ defining the scalar field of our d-dimensional mesh is continuous.
2. All the cells of the mesh are connected.
3. A function $R(c)$ is given which, for any given cell c of the mesh, returns the *range* of values assumed by $\mathcal{F}$ over the domain of c. Note that, since $\mathcal{F}$ is continuous, the range returned is always an interval $[min_c, max_c]$.
4. For each pair of adjacent cells (c_i, c_j), let

$$f_{ij} = \{\mathbf{x} | \mathbf{x} \in c_i, \mathbf{x} \in c_j\}$$

and define the *connecting interval*:

$$R(f_{ij}) = \left[\min_{\substack{\mathbf{x}\in c_i \\ \mathbf{x}\in c_j}} \mathcal{F}(\mathbf{x}), \max_{\substack{\mathbf{x}\in c_i \\ \mathbf{x}\in c_j}} \mathcal{F}(\mathbf{x})\right] \subseteq R(c_i) \cap R(c_j)$$

such that if the cell $c_i(c_j)$ is processed for a value $w \in R(f_{ij})$, then the cell $c_j(c_i)$ will be also processed for the same value w. This is essentially the information we get from the contour propagation scheme.

Based on the above assumptions, we construct a labeled graph G. Note that this graph need not be constructed explicitly in practice. For each cell c in the mesh, we have a node n_c in G which is labeled $\mathrm{T}(n_c) = R(c)$. For each pair of adjacent cells (c_i, c_j), there is an arc f_{ij} in G connecting n_{c_i} to n_{c_j} which is labeled $\mathrm{T}(f_{ij}) = R(f_{ij})$. The arc f_{ij} corresponds to the *face* which is shared by cells c_i and c_j.

Connectivity relations between nodes in the graph G are transfered to relations between the corresponding cells of the underlying mesh. Based on propagation of contours through cell adjacencies we have the following definition:

Definition 1 *Consider a scalar value w and a connected sequence of nodes*

$$\mathcal{P} = \{n_{i_1} \cdots n_{i_k}\}$$

*$\mathcal{P}$ is called a w-*path *if*

$$w \in R(f_{i_j i_{j+1}}), \forall j \in [1 \cdots k-1]$$

A w-path represents a cell traversal sequence based on application of a contour propagation algorithm for a given isovalue w. We further define:

Definition 2 *Consider a scalar value w and two nodes n_{c_i}, n_{c_j} of G. n_{c_i} and n_{c_j} are said to be w-*connected *if there exists a w-path connecting them.*

Note that Definition 2 is a transitive relation, and we can define:

Definition 3 *A maximal set of nodes $\{n_{i_1} \cdots n_{i_k}\}$ which are w-connected is called a w-connected component.*

A w-connected component defines precisely the set of cells which are processed by contour propagation from a single cell in the set. Note that a w-connected component differs slightly from a *connected component* of the isocontour, in that two separate connected components may intersect a common cell, forming a single w-connected component, as illustrated in Figure 10.

We can extend the concept of w-connectivity between pairs of cells to the connectivity of a set of cells with respect to a range of values.

Definition 4 *Consider a subset S of the nodes of G and a node $c \in G$. The node c is* connected *to S if, for any $w \in R(c)$, there exists a node $c' \in S$ that is w-connected to c.*

2.2 Seed Sets

We now characterize some particular subsets, called *seed sets*, of the cells of a mesh in terms of the connectivity properties defined in the preceding subsection. Seed sets are important because all connected components of any isocontour of the entire original mesh can be traced by contour propagating from the cells of any seed set.

Definition 5 *A subset S of the nodes of G is a* seed set *of G if all the nodes of G are connected to S.*

In order to quickly determine all cells whose range contains a particular scalar value w, we can proceed as follows:

1. search for all the cells $c \in S$ such that $w \in T(c)$;

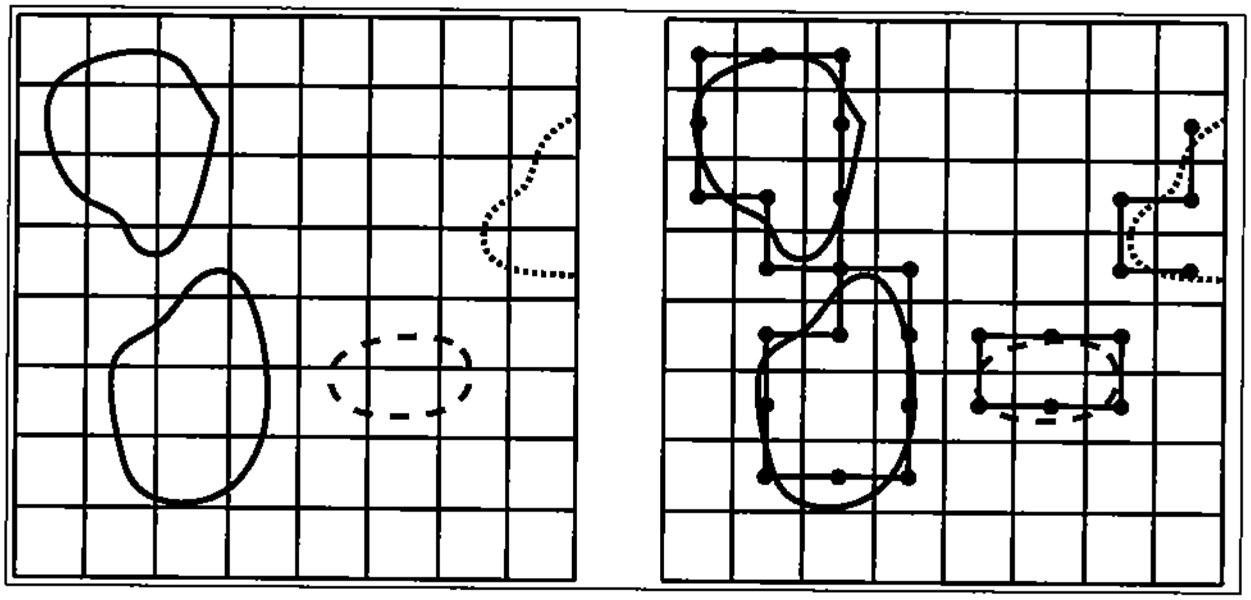


Figure 10: Illustration of w-connected components. On the left are four contour components for a particular isovalue w. On the right a portion of the graph G is displayed, corresponding to the three w-connected components. Displayed in green is a w-path between two of the nodes.

2. starting from the cells reported in step 1 and using the w-connectivity relation on the graph G (that is the contour propagation scheme), we find all the cells of the mesh whose range contains w.

To reduce the search time and storage requirements it is desirable to reduce the cardinality of the seed set S as much as possible. Toward this end we will apply the following property:

Property 1 *If S is a seed set and $c \in S$ is a cell connected to $S - \{c\}$, then $S - \{c\}$ is a seed set.*

Proof: By hypothesis we have that c is connected to $S - \{c\}$. Also, from Definition 2, we have that any cell which is w-connected to c is also w-connected to some cell in $S - \{c\}$. Hence $S - \{c\}$ is a seed set. ◇

Property 1 provides us with a method to reduce the size of a seed set. If we wish to find a small seed set, we can start with the entire set of the cells – that is the largest seed set – and keep removing cells until we achieve a *minimal seed set*. Note that a minimal seed set is not the seed set with the minimum number of cells but a seed set from which we cannot remove any cell to obtain a new seed set.

The repeated application of Property 1 requires the knowledge at each step of the connectivity relations within the current seed set. Thus, we may start from the initial graph G. At each step, we remove a selected cell c along with all its incident arcs and add some new arcs between pairs of cells that were connected to c to take into account the connectivity relations induced by c on $G - \{c\}$. In particular, if two cells c_i and c_j are both connected to c_k with arcs f_{ik} and f_{jk}, then the removal of c_k requires also the removal of f_{ik} and f_{jk} and potentially the insertion of a new arc f_{ij} connecting c_i to c_j. This new arc f_{ij} needs to be inserted if $R(f_i) \cap R(f_j) \neq \emptyset$ (a case in which the transitivity of Definition 2 applies). If this condition is true, then the new arc is added with label $R(f_{ij}) = R(f_i) \cap R(f_j)$. If we proceed in this way, it becomes simple to determine if Property 1 can be applied. We can remove a cell c_k of the current seed set if:

$$\bigcup_{i=1}^{k} R(f_{ik}) = R(c_k)$$

where $f_1, \ldots, f_j$ are all the arcs incident to the cell c_k in the reduced graph of the current seed set.

Given this general reduction scheme, we still have freedom to select the cells to be removed in any order. We can use a greedy approach, removing first the cells that we consider less likely to belong to a minimal seed set – for example the cells that have narrower range. In this way we can assume that the minimal seed set we achieve is not much larger than the seed set with the minimum number of cells. On the other hand, we can use this freedom to make the algorithm as simple as possible (a very important property in actual implementations).

In the following subsections we present three seed selection algorithms. In Section 2.3 we present a greedy approach for constructing near-optimal seed sets for irregular or regular grids. In Section 2.4 we present a simple, fast seed selection approach by a sweep traversal for both irregular and regular grids. In a regular grid the sweep process can be simply implemented as a traversal of the grid by rows using a regular marching scheme. In Section 2.5 we examine a modified case for grids of regular topology which achieves smaller seeds sets with slightly larger temporary storage complexity.

In our seed set generation algorithms, we begin by considering the *universal seed set*, consisting of all cells in the mesh. We associate with each seed cell a computed range $\mathrm{T}(c) \subseteq R(c)$, which represents the range of values for which the given cell is a seed cell. Initially, we have $\mathrm{T}(c) = R(c)$, the entire range of the cell, hence S is trivially a seed seed set. Algorithms for seed set generation can be viewed as seed set reduction techniques, which iteratively apply Property 1 to the current seed set to reduce the size of S.

2.3 Greedy Climbing

For computation of a nearly optimal seed sets we develop a greedy technique which progressively covers the domain with seed cells by explicitly computing the coverage of each seed cell introduced. This *climbing* algorithm can be applied to any complex of cells provided that the appropriate function R is given which computes the range of a cell or face. The main advantage of the seed cell selection algorithm presented in [31] is that it guarantees the computation of a seed set nearly optimal in size (at most twice the size of the optimal seed set). What makes it difficult to achieve such a goal is the problem of selecting the "best" seed cells at the saddle points of the scalar filed. The cost of solving such "difficult" situations is:

- the necessity to build explicitly the contour tree [31] of the scalar field;
- the use of complex data structures as the union-find-split (generalization of the standard union-find);
- the use of involved routines like the tandem-search.

As a consequence it becomes difficult to use such an approach in practice even if the results it produces are highly desirable. Moreover we observe that in many scalar fields which arise in practice the number of saddle points is relatively small and, more important, a greedy selection (not based on contour tree and tandem search) gives in practice very good results. As a consequence we have developed a "practical" algorithm which computes nearly optimal seed sets (even if not guaranteed in theory) for general unstructured meshes. The generalization to the unstructured grids is based on the application of:

- greedy seed cell selection
- contour driven advancing front

We begin by considering the universal seed set S with $\mathrm{T}(c) = R(c)$ for all cells c. In our algorithm processing begins by selecting an arbitrary cell c. For the example in Figure 11 the first seed cell is the lower left cell of the grid (in practice the quality of the result is not going to be greatly affected by the selection of the first cell). We can trace as in Figure 11(a) the isocontour $C(i)$ which bounds the range of the cell c. The cells intersected or included within the isocontour $C(i)$ are processed so that their range is reduced only to the portion outside i, as in Figure 11(b). At each step the two following operations are performed:

(i) Among the cells across the current front i we select as the next cell to process the one with larger residual range $(\mathrm{T}(c) - R(c))$.

(ii) The front i is enlarged up to the isocontour that includes the newly selected seed cell.

The geometric interpretation of the algorithm is based on the idea of contour tree [31] (note that we use the contour tree as algorithm analysis tool, but we do not need to build it). The greedy choice is equivalent to selecting each time the seed cell that allows to climb (descend) as fast as possible the contour tree (see Figure 11(c)(d)). To achieve this without computing the contour tree we need to resort to the contour driven advancing front. The contour front is realized using a priority queue in which we store the cells that are intersected by (but not included in) the current front i. As the front i advances new cells are inserted in the priority queue while other are removed (those included in the new advanced front). To access and efficiently update the cells within the priority queue we compound a hashing scheme that allows access in average expected $O(1)$ time to any cell in the queue. The base algorithm consists of:

```
ContourClimbing($mesh$)
    select an initial cell $c$
    insert $c$ in the priority queue $p$ with priority $R(c)$
    while $p$ is not empty
```

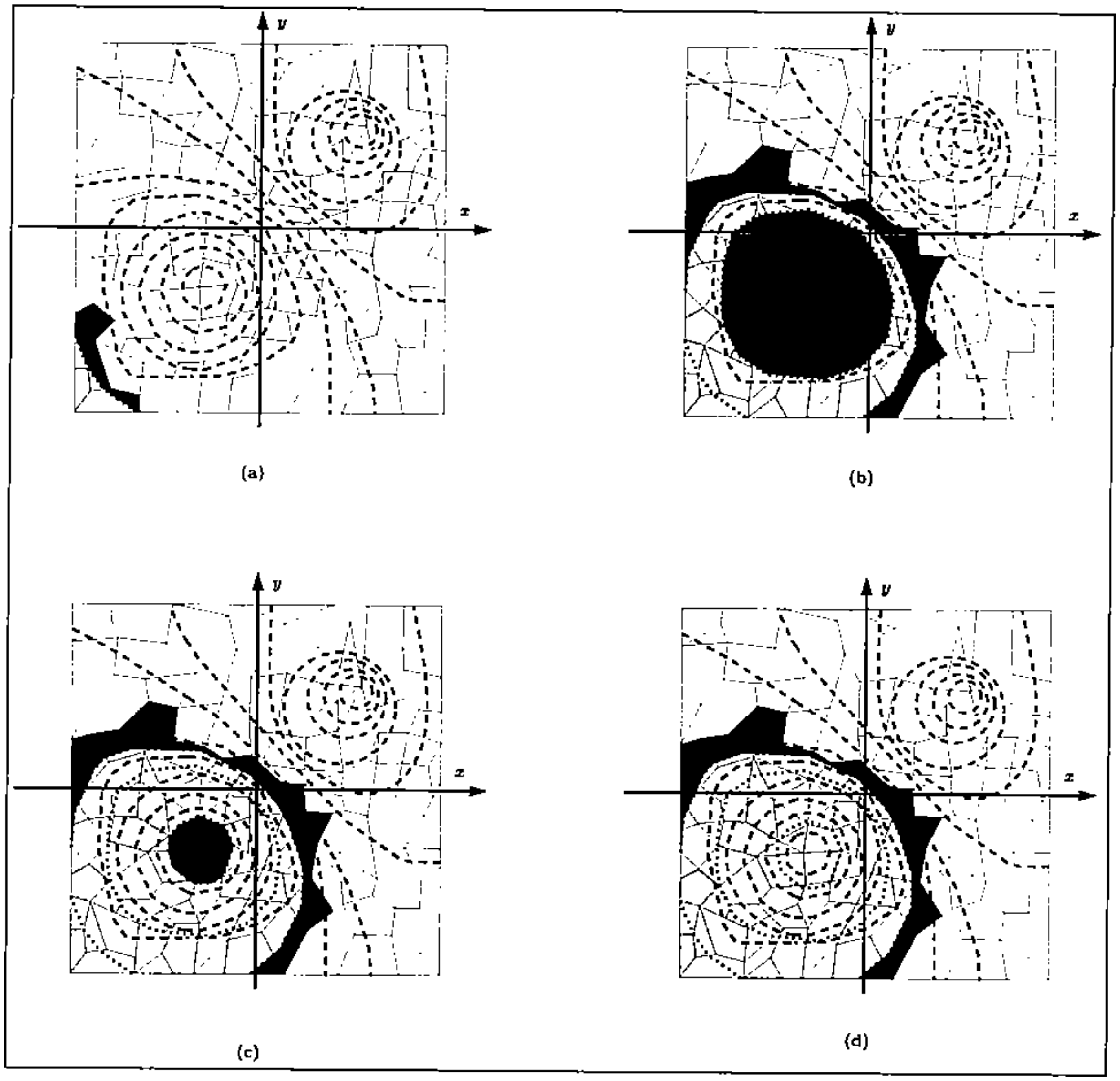


Figure 11: Greedy climbing approach to seed cell selection. Grey cells represent the selected seed cells. Yellow cells have been processed and removed from consideration, while red cells represent the current *front* of cells from which the next seed cell will be chosen.

```
do
    extract the cell $c$ with highest priority and associated range $\mathtt{T}(c)$
    { cell $c$ with range $\mathtt{T}(c)$ remains in the seed set }
    PropagateRegion($mesh$, $c$, $\mathtt{T}(c)$, $p$)
done
```

The function PropagateRegion() is an extension of contour propagation to the case of simultaneous propagation of an *interval* of values. Similar to contour propagation, interval propagation uses a queue of cells and propagates from cell to adjacent cell. Associated with each cell c_i in the queue is an interval $P(c_i)$, which represents the interval range which has been propagated to cell c_i. When intervals are passed from a cell c_i to a neighboring cell c_j, only values in the shared range $R(f_{ij})$ can be propagated, as the purpose is to mimic contour propagation for a range of values. The overall interval propagation algorithm is outlined as follows:

```
PropagateRegion($mesh$, $c$, $\mathtt{T}(c)$, $p$)
    insert $c$ in the queue $q$ with associated range $\mathtt{T}(c)$
    while $q$ is not empty
    do
        extract the cell $c_i$ and associated range $P(c_i)$ from $q$
        if cell $c_i$ is in $p$
        then
            $\mathtt{T}(c_i) = \mathtt{T}(c_i) - P(c_i)$
            if $\mathtt{T}(c_i) = \emptyset$
            then
                remove cell $c_i$ from $p$
            else
                set priority of $c_i$ to the new span of $\mathtt{T}(c_i)$
            endif
        endif
        for each cell $c_j$ adjacent to $c_i$
        do

            if $c_j$ is in the queue $q$
            then
                $P(c_j) = P(c_j) + P(c_i) \cap R(f_{ij})$
            else
                add $c_j$ to $q$ with associated range $P(c_j) = P(c_i) \cap R(f_{ij})$
        done
    done
```

Figure 12 shows a sample seed set computed with the algorithm described above.

2.4 Sweep Filtering

Computation of seed sets need only be performed one time for any dataset, and the results can be stored off-line. In many cases a considerable amount of processing can be devoted to generating very small seed sets, and the results can be used over and over again. However, in many practical situations, the maintenance of a priority queue as described in the contour climbing algorithm may be prohibitively expensive. Such situations include:

1. time critical – The time complexity of the contour climbing algorithm may be prohibitive if results are needed very quickly.
2. dynamic data – If the data are being collected an analyzed in real-time, contour climbing may be infeasible.
3. out-of-core – If the data are too large to be stored in main memory, a memory access pattern which exhibits greater coherence than the random-access pattern of contour climbing would be desirable.

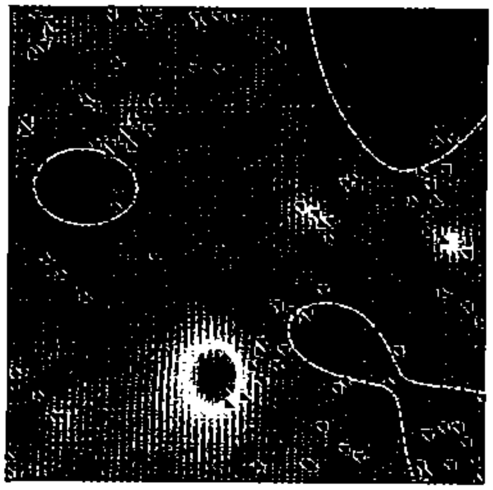


Figure 12: Results of seed selection by contour climbing (76/7938 cells)

We present a simple seed selection algorithm which is motivated by these practical considerations. The seed selection is conceptually easiest to understand as a sweep of the cells in a particular direction. The algorithm has the property that selected seeds fall on the extrema of the contours in the given sweep direction. Detection of contour extrema is based on a simple comparison of the gradient within each cell and its immediate neighbors. With such a seed set, contouring may be performed coherently and efficiently by executing a contouring sweep, with only a slice of data required to be resident in memory at any given time, resulting in efficient computation for visualization of large out-of-core datasets.

To understand the properties that such a seed set must have, we first consider the one pass contour tracing algorithm. From its analysis we immediately observe the properties that an appropriate seed set must have.

2.4.1 One-pass Contour Tracing

Conceptually the one-pass contour tracing is based on a sweep of the cells along a particular direction. As illustrated in Figure 13, a sweep line l (sweep plane in 3D) is moved from left to right along the x direction. The isocontour $C(w)$ of height w is built progressively as it is crossed by the sweep line l. Each time l is tangent to the isocontour $C(w)$ three situations may arise:

- $C(w)$ attains a local minimum along the direction $\vec{l}_\perp$ (orthogonal to l) as in Figure 14(a)-(b). A new portion of $C(w)$ starts to be traced.
- $C(w)$ attains a local maximum along the direction $\vec{l}_\perp$ as in Figure 14(c)-(d). Two separate portions of $C(w)$ may join or a loop may be closed.
- An inflection point is met that does not need any special processing, as in Figure 14(e).

To perform the contour sweep operation two conditions suffice:

- The dataset is stored with the cells sorted by maximum $\vec{l}_\perp$ so that by loading them into memory (from the end to the beginning) we automatically perform a sweep. This means that in the contour tracing stage and in the seed selection stage the sweep algorithm can be performed in linear time.
- Any local maximum along the $\vec{l}_\perp$ direction of any isocontour can be immediately detected (that is, a proper seed set is precomputed). This allows us to avoid loading in memory sections of the mesh where there is no contour component.

2.4.2 Sweeping Seed Selection

The seed selection stage is performed with a forward sweep as illustrated in Figure 15. Conceptually, the sweep line l is moved from left to right to determine the order in which cells are processed. Note that this ordering is not required by the selection algorithm, and so cells which are stored in main memory can be processed in any order, or even in parallel. Since the cells are already sorted in the mesh, the sweep is achieved by simply loading them into memory in the order they are stored. When a cell c is met which contains a local maximum of an isocontour along the sweep direction $\vec{l}_\perp$ the cell c is added to the seed set.

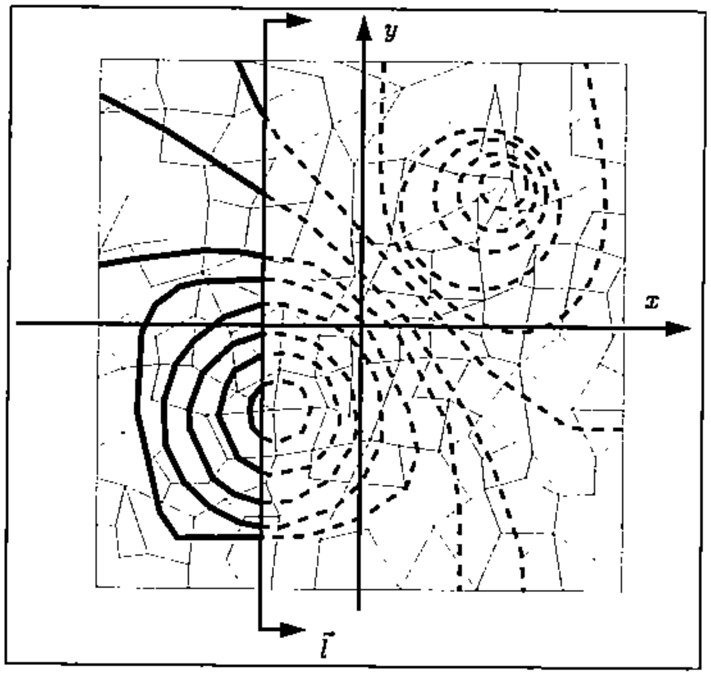


Figure 13: Illustration of one-pass contour sweeping. Contour components are computed as they are crossed by the sweep line.

From the geometrical point of view the determination of a local maximum within a cell of an unstructured mesh is based on the normal $\vec{m}_c$ of the contour within the cell c. Consider two adjacent cells c and d, as shown in Figure 16. The normal $\vec{m}_c$ ($\vec{m}_d$) of the contours within c (d) is the projection onto the mesh space of the normal of the scalar field in c (d). As in Figure 16 we can determine a fixed direction $\vec{l}$ along l and perform the following test:

- if both $\vec{m}_c$ and $\vec{m}_d$ have positive (negative) scalar product with $\vec{l}$ then neither a maximum nor a minimum is met;
- Assume that $\vec{m}_c \cdot \vec{l} \geq 0$ and $\vec{m}_d \cdot \vec{l} \leq 0$ (the opposite case is symmetric). If the cross product $\vec{m}_c \times \vec{m}_d$ has positive z direction then a minimum is met.

Note that in order to test whether a cell is a seed it is only necessary to examine the cell and its first neighboring cells. For data stored in primary memory, this is not an issue. For the out-of-core extension to this approach, we must ensure that when a cell is tested, all neighboring cells are available in primary memory as well. There are essentially two approaches to solving this. In the first approach, we can determine the maximum difference between the indices of adjacent cells, in order to compute the amount of primary memory necessary for the out-of-core processing. A more attractive solution is to fix the amount of memory which is available for the out-of-core processing, and adapt the seed selection to this limit. In this case, if an adjacent cell is not available in primary memory during the seed selection, the shared face between the cells is treated as a *border* of the mesh. Applying this approach, we may select more seed cells, with the advantage that the approach can adapt to situations in which only a small fraction of the mesh can be stored in primary memory at any given time.

Sweep filtering requires $O(n_c)$ time for considering each cell, and no additional storage beyond that of the extracted seed set (and the portion of the mesh kept in memory).

Note that in addition to facilitating out-of-core computation, in general the sweep filtering approach provides an extremely efficient method for computing a small seed set. Moreover, due to the local criteria for seed selection, cells may be considered in any order, allowing for parallel implementation with little or no communication overhead during the preprocessing.

2.4.3 Special Cases

While the algorithm described above is general and independent of grid topology, special considerations for particular types of data and grids may be worthwhile. The criteria given above for detecting a minima along the sweep direction have a special condition in the case that $\vec{m}_c \cdot \vec{l} = 0$ and $\vec{m}_d \cdot \vec{l} = 0$, indicating that the contour is

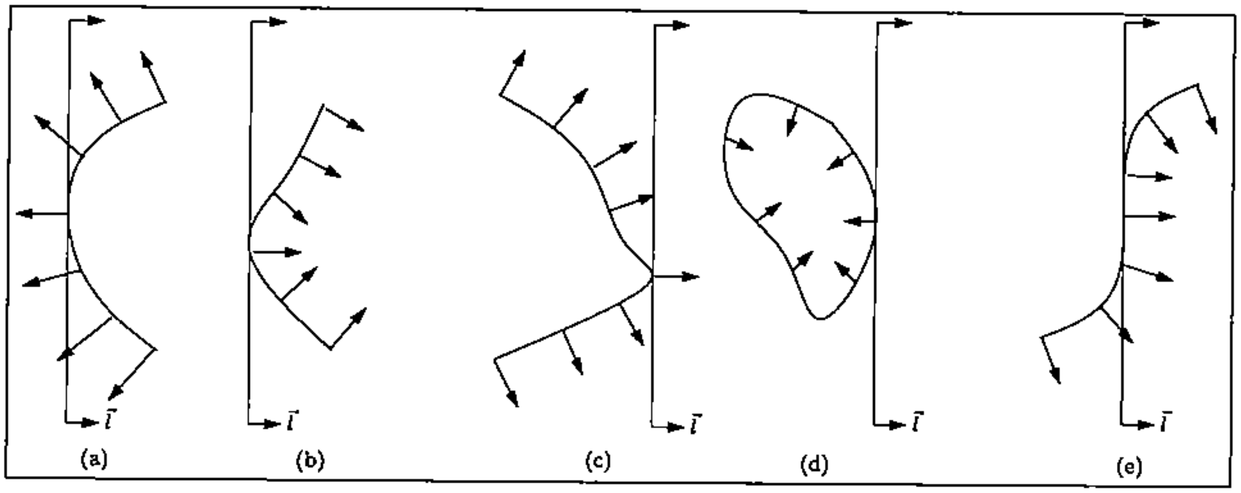


Figure 14: Tangent conditions of contour with sweep direction

perpendicular to the sweep direction. While this special case may not occur frequently in a general data setting, the frequency of occurrence is much greater in particular settings, such as integer-valued data defined over a regular grid, which is often the case for digital terrain data and medical image data.

With such data, it is not uncommon for degenerate situations to occur, as illustrated in Figure 18. In this case, a minimum along the sweep direction is an entire line, which may result in a *column* of seed cells for a particular isovalue, though it is clear that the seed cells are w-connected. In higher dimensions, the problem remains that a large number of cells along the $d-1$ dimensional hyperplane may be selected.

This degenerate situation is easily and efficiently addressed by making slight modifications to the selection criteria. By modifying the minima detection criteria that $\vec{m}_c \cdot \vec{l} \geq 0$ and $\vec{m}_d \cdot \vec{l} < 0$, (one perpendicular condition is removed), only one seed along each flat minima region will be chosen, as illustrated in Figure 19. For regular grids of higher dimension, similar consideration of special cases can be made.

A sample seed set computed by sweep filtering is shown in Figure 20.

2.5 Responsibility Propagation

The special case for regular grids may be extended to provide smaller seed sets with a constant increase in computational complexity and only slightly greater storage. This technique can be viewed as a simplification of the connectivity graph technique described in the Section 2.1 for determining a seed set S. The algorithm does not require that we store the entire graph, but instead we maintain a subset of the information from the graph which can be locally propagated from cell to cell using simple rules when marching in a regular order, with temporary storage complexity of $O(n^{(d-1)/d})$. We again begin with all cells c in the set S. We associate with each seed cell a computed range $\mathtt{T}(c)$, which represents the range of values for which the given cell is a seed cell. Initially, we have $\mathtt{T}(c) = R(c)$, the entire range of the cell, hence S is a trivially a seed seed set. We present an incremental seed elimination technique to reduce the seed set S. The reduction and removal of seed cells is based on propagation of *responsibility ranges* of isovalues. The information propagated from cell to cell in marching order is a range $\mathtt{T}$ for each dimension of the regular grid. An *incoming range* $\mathtt{T}$ represents the range of values w for which responsibility has been propagated to the current cell from the neighboring cells. The incoming range is always a subset of the range of the shared face in the direction of propagation. The complement of the incoming range in the direction which varies fastest consists of values w for which the current cell is w-connected to either (i) a processed cell which remains in the seed set or (ii) an unprocessed cell to which responsibility for the value w has been propagated. An *outgoing range* represents the responsibility range which is propagated from the current cell to a neighboring cell. Illustrated for the 2D case in Figure 21, the marching order is Y varying fastest, X varying slowest.

We describe the processing of a cell c at index (i, j) in a topologically regular grid of dimension (n_x, n_y). Boundary conditions are handled directly through the following notation, defined for simplicity:

1. $\mathtt{T}(f_u)$ represents the range of the incoming face in the U direction, where U is an arbitrary dimension.
2. $\mathtt{T}(u)$ represents the incoming range propagated in the U direction. In the case of the boundary condition $u = 0$, we take $\mathtt{T}(u) = \mathtt{T}(f_u)$.

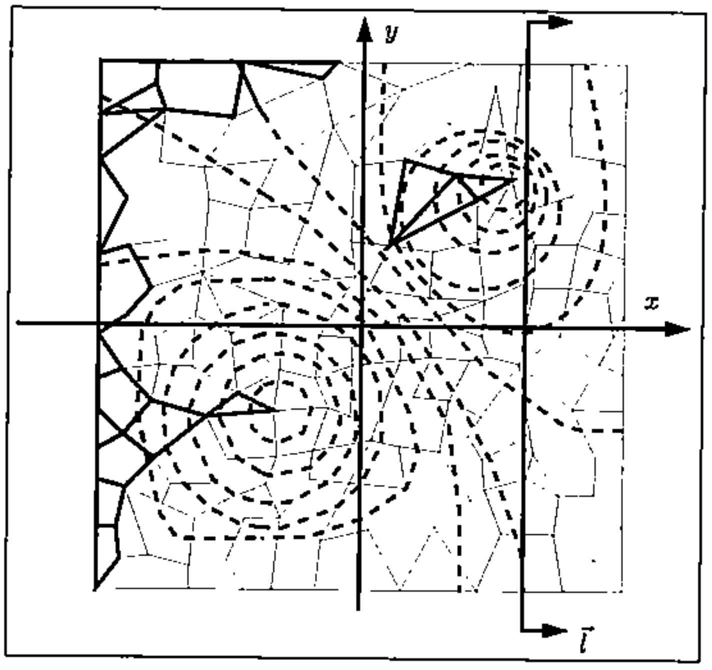


Figure 15: One-pass seed selection by forward sweep

3. $\overline{T(u)}$ represents the complement of $T(u)$ with respect to the range $T(f_u)$ of the shared face, or $T(f_u) - T(u)$. Note that the propagated range $T(u) \subset T(f_u)$.
4. $T(f_{u'})$ represents the range of the outgoing shared face in the U direction. In the boundary case when there is no adjacent cell in the outgoing U direction ($u = n_u - 2$), we assign $T(f_{u'}) = \emptyset$, indicating that no propagation may occur in the given direction.
5. $T(u')$ represents the range propagated from the current cell to the outgoing adjacent cell in the U direction.

We first compute the combined incoming range $T(I)$, and corresponding complement range $\overline{T(I)}$:

$$T(I) = (T(y) \cup T(x)) - \overline{T(y)} \tag{1}$$

$$\overline{T(I)} = (T(f_x) \cup T(f_y)) - T(I) \tag{2}$$

$T(I)$ represents the subset of incoming isovalues which cell c must either account for in the seed set S or defer responsibility for by propagation through $T(x')$ and $T(y')$. The removal of $\overline{T(y)}$ in Equation 1 above is justified based on the algorithm for range propagation presented below. For all $w \in \overline{T(I)}$, there either exists a processed cell in S which is w-connected to c or the value w has already been further propagated, and hence $w \in \overline{T(I)}$ need not be considered in processing c. This leads to the definition of $T(R)$, representing the entire range of values which make up the responsibility range of cell c.

$$T(R) = R(c) - \overline{T(I)} \tag{3}$$

For $w \in T(R)$, we must take care that c remains w-connected to S in order to maintain the property that S is a seed set. We also compute $T(P)$, which represents the combined range of isovalues which may be further propagated through outgoing faces:

$$T(P) = T(f_{x'}) \cup T(f_{y'}) \tag{4}$$

We arrive at the following greedy algorithm for deferring seed cell selection through propagation of responsibility. Through the processing of a cell c, we maintain the invariant that S is a seed set.

```
if ($T(R) \subseteq T(P)$) then
    { Cell c can be removed from S }
    $S = S - c$
    { Propagate responsibility ranges }
    $T(x') = T(f_{x'}) \cap T(R)$
    $T(y') = T(f_{y'}) \cap (T(R) - T(x'))$
```

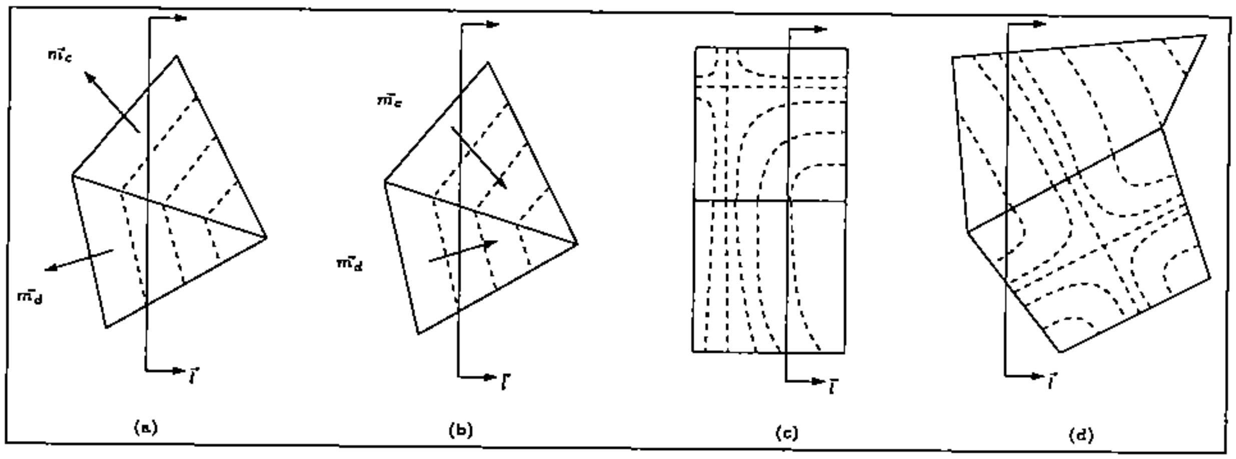


Figure 16: Conditions for determining a local maxima along the $\vec{l}_\perp$ direction. In linear cells (a-b) the maxima lie along cell edges. With regular cells (c) the maxima remain along edges, though the the conditions may change along the length of the edge. In non-linear cells as simple as the non-axis-aligned bi-linear cell (d), maxima may occur in cell interiors.

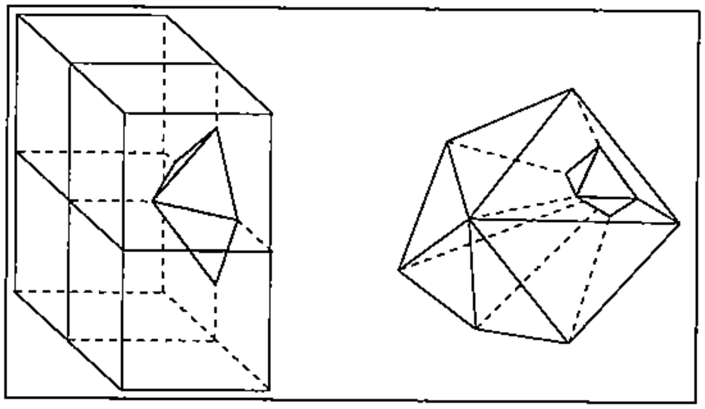


Figure 17: Three dimensional examples of local minima along the sweep direction.

```
else
    { Cell c must remain in the seed set }
    $\mathrm{T}(c) = \mathrm{T}(R)$
    $\mathrm{T}(x') = \emptyset$
    $\mathrm{T}(y') = \emptyset$
end
```

Proof: (S remains a seed set after processing of cell c)

Case 1 ($\mathrm{T}(R) \subseteq \mathrm{T}(P)$) - Recall that cell c is w-connected to a processed seed cell for $w \in \overline{\mathrm{T}(I)}$. Through propagated responsibility ranges, we have that c is w-connected to the remaining (unprocessed) seed set for $w \in \mathrm{T}(x') \cup \mathrm{T}(y') = [\mathrm{T}(f_{x'}) \cap \mathrm{T}(R)] \cup [\mathrm{T}(f_{y'}) \cap (\mathrm{T}(R) - \mathrm{T}(x'))] = (\mathrm{T}(f_{x'}) \cup \mathrm{T}(f_{y'})) \cap \mathrm{T}(R) = \mathrm{T}(P) \cap \mathrm{T}(R) = \mathrm{T}(R) = R(c) - \overline{\mathrm{T}(I)}$. Thus, c is connected to $S - \{c\}$, and by Property 1, $S - \{c\}$ is also a seed set, maintaining the invariant property.

Case 2 (Cell c remains in the seed set) - Cell c is trivially w-connected to S for $w \in \mathrm{T}(c) = \mathrm{T}(R) = R(c) - \overline{\mathrm{T}(I)}$. From the input conditions, we have that c is w-connected to a processed cell which remains in S for $w \in \overline{\mathrm{T}(I)}$. Thus, c is w-connected to S for $w \in R(c)$, maintaining the invariant property that S is a seed set.

◇

In the first case, the propagated range $\mathrm{T}(P)$ includes the responsibility range $\mathrm{T}(R)$ in its entirety, and cell c is removed from the seed set S. The responsibility range is propagated through the outgoing faces by the

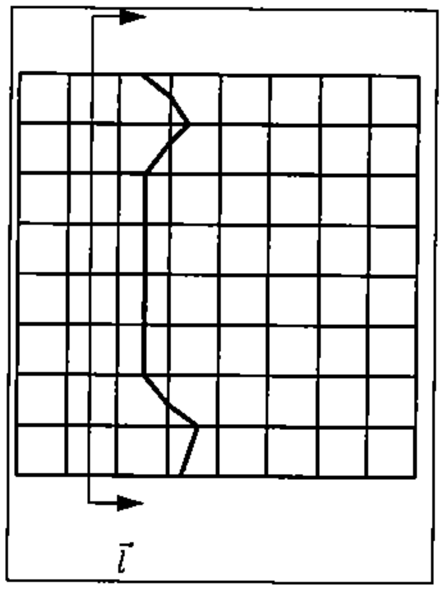


Figure 18: Degenerate minima which occur with greater frequency in grids of regular topology with integer-valued data

computation of $T(x')$ and $T(y')$. Note that the propagated ranges are disjoint and that the preference is to propagate the range in the X direction. It is this preference which allows us to remove $\overline{T(y)}$ in equation (1). For all $w \in \overline{T(y)}$, the associated w-connected component is either accounted for by a processed cell in the seed set S, or responsibility has been propagated to an unprocessed cell, hence w need not be considered for the current cell. The same cannot be said for $\overline{T(x)}$, because the precedence of propagation indicates that responsibility for values $w \in \overline{T(x)}$ may, through some path of responsibility propagation, ultimately be propagated through $T(y)$. Consider the case of Figure 21, and suppose that the value A is a local minimum. Values $w \in \overline{T(x)}$ overlap with the range $T(y)$, providing incoming information which appears to conflict. In fact we cannot make use of the range $\overline{T(u)}$, where u is other than the direction which varies fastest in the marching order.

The second case above occurs when cell c cannot propagate the entire incoming range. Cell c remains in the set S, though $T(c)$ is reduced to exclude the complement ranges which have been propagated elsewhere. In this case the empty set is propagated to outgoing edges, indicating that all values on shared faces are accounted for in the seed set S.

As described above, the *range propagation* method for selecting seed cells requires $O(n^{(d-1)/d})$ storage to maintain the propagated ranges for a sweeping line or plane in 2D or 3D. Note that our use of range subtraction may result in ranges with two disconnected components. In practice, disconnected ranges may either be maintained or closed by taking the smallest range which contains the entire disconnected range. Maintaining the disconnected range effectively requires that multiple seeds be processed into the search structure, increasing the number of seeds, while merging disconnected ranges simply means that two or more cells which are w-connected may be selected for inclusion in the seed set S. Of course, this greedy technique does not guarantee the selection of a single cell for each connected component in the case that disconnected ranges are maintained. In our implementation, we maintain disconnected ranges through the seed cell selection, closing each range which is ultimately selected to remain in the seed set S. In practice the number of seed cells with disconnected ranges does not exceed 10% of the seed cells, and the number of seed cells does not exceed 10% of the data, as presented in the results.

Results for a our 2D sample function are illustrated in Figure 22. The relatively smooth function is sampled on a grid of size 64×64. Figure 22 shows the 206 seed cells chosen by the range propagation seed selection method.

2.6 Seed Set Results

Table 1 presents the comparative sizes of seed sets for the three seed selection algorithms applied to a variety of input 2D and 3D meshes.

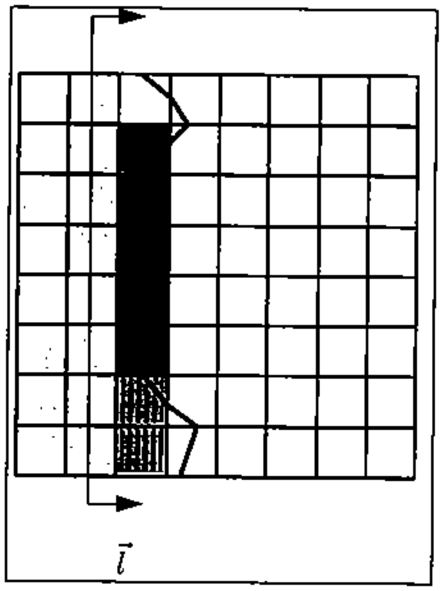


Figure 19: The cells on the current sweep plane are processed in regular order. A bit flag is turned *on* when a local maximum exists on the top edge of a cell.

Dataset	Climbing	Propagation	Sweeping	Checker	Total Cells
Eagle Pass Terrain	1872	7151	29356	720000	1440000
Sample Function	59	177	238	1985	3969
Hipip	529	2212	6397	62559	250047
Climate Data	177	602	1916	4760	19040
SOD	2308	9944	18608	264960	1059840

Table 1: Seed set sizes for the three presented algorithms, compared with the checkerboard approach and the total number of cells.

3 Range Queries

The fundamental isocontouring query concerns the enumeration of all cells c such that $w \in R(c)$ for the input isovalue w. In this section, we compare the use of three data structures supporting this *range query* operation in terms of the storage complexity and the time complexity for both creation of the structure and for performing individual queries. While the characteristics of the search structures being studied are easily understood and compared in theory, characteristics of our data and seed sets lead us to examine the practical application considerations. A primary consideration is that of the data type. Note that for integer-valued data, the search structures listed below all simplify to the same complexity, both in space and query time. A second consideration is the size of the seed sets. While in the worst case $n_s = O(n_c)$ (note that we always have that $n_s \leq n_c$ as S is a subset of all cells), we have demonstrated in practice that n_s is often smaller than n_c by one or more orders of magnitude. This leads us to consider search structures of greater space complexity, which may lead to improved query complexity or practical demonstrated performance.

In the following sections we review the *interval tree*, *segment tree*, and *bucket* search structures as applied to the contour query problem described. Example search structures are illustrated for the input set of intervals shown in Figure 23. For each search structure, we describe:

- Data structures
  Basic C/C++ data structures for representing the search structure
- Construction
  The algorithm for creating a search structure from a set of input intervals. The search structures are not created iteratively, and so balancing of the trees is not an issue.
- Querying
  The algorithm for processing an input query for a given isovalue w is considered.

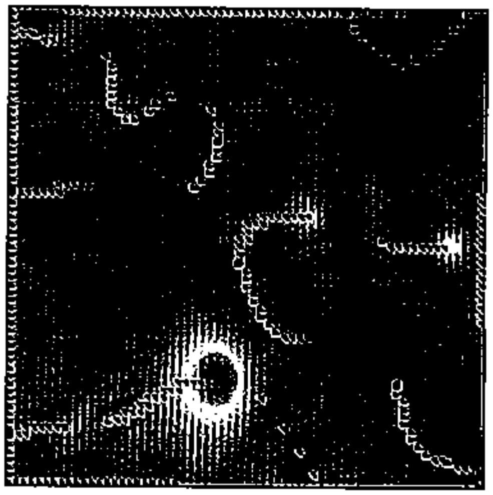
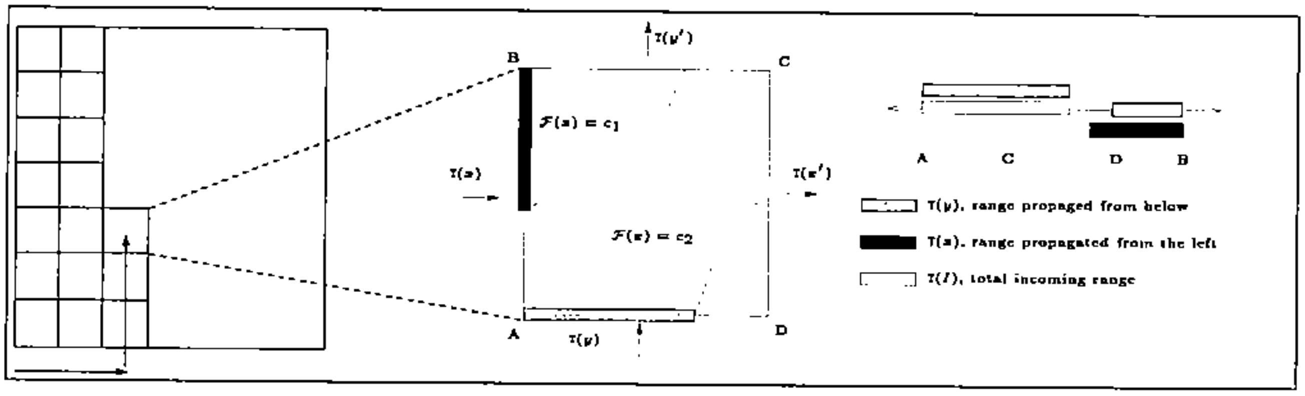


Figure 20: Results of seed selection by directional sweep (296/7938 cells)



B
C
A
D
$\mathcal{T}(y')$
$\mathcal{F}(x) = c_1$
$\mathcal{T}(x)$
$\mathcal{T}(x')$
$\mathcal{F}(x) = c_2$
$\mathcal{T}(y)$
A
C
D
B
$\mathcal{T}(y)$, range propaged from below
$\mathcal{T}(x)$, range propagated from the left
$\mathcal{T}(I)$, total incoming range

Figure 21: Illustration of responsibility propagation. Each cell processes input responsibilities and produces output responsibilities

Our analysis and data structures are based on the general definitions of the data structures, without respect for the data from which they are derived. In particular, the search structures are designed such that queries can be resolved without referencing the original data.

3.1 Interval Tree

An *interval tree* is made up of a binary tree over the set of interval *min/max* values [19]. Each internal node holds a *split value s*, with which intervals are compared during insertion into the tree. If the interval is entirely *less than* the split value it is inserted into the left subtree, while intervals *greater than* the split value are recursively inserted into the right subtree.

In the case that the interval spans the split value ($min < s < max$), the recursion terminates and the given interval is stored at the current node. Each nodes maintains two list of spanning cells. The first list is stored in increasing order by the *min*, the second in decreasing order by the *max* value. Because the intervals are not split in the recursive insertion, each interval is stored only twice, and the storage complexity is $O(n_s)$.

3.1.1 Interval Tree Data Structure

The data structure for the interval tree is relatively simple. The tree structure is implicit, with no need for pointers (in the case that the intervals are static).

```
struct IT_IntervalList {
```

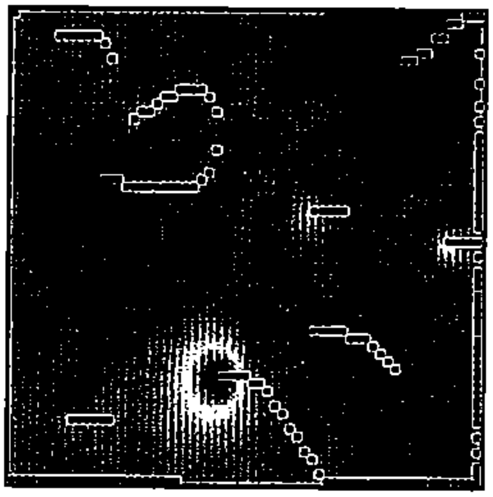


Figure 22: Results of seed selection by range propagation (206/3969 cells)

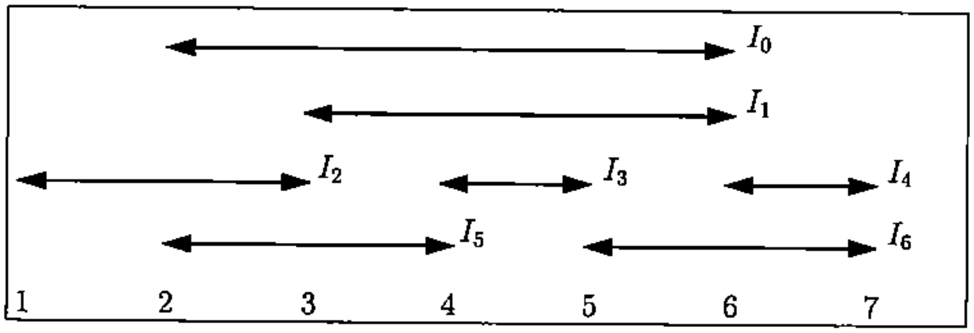


Figure 23: A set of segments representing cell ranges

```
   int *interval_id;
};

struct IT_Node {
   float        split_value;
   int          n_intervals;
   IT_IntervalList min_list;
   IT_IntervalList max_list;
};

struct IT_Interval {
   float min;
   float max;
   int cell_id;
};

struct IntervalTree {
   int          n_intervals;
   IT_Interval *intervals;

   int          n_uniq_val;
   IT_Node     *nodes;
};
```

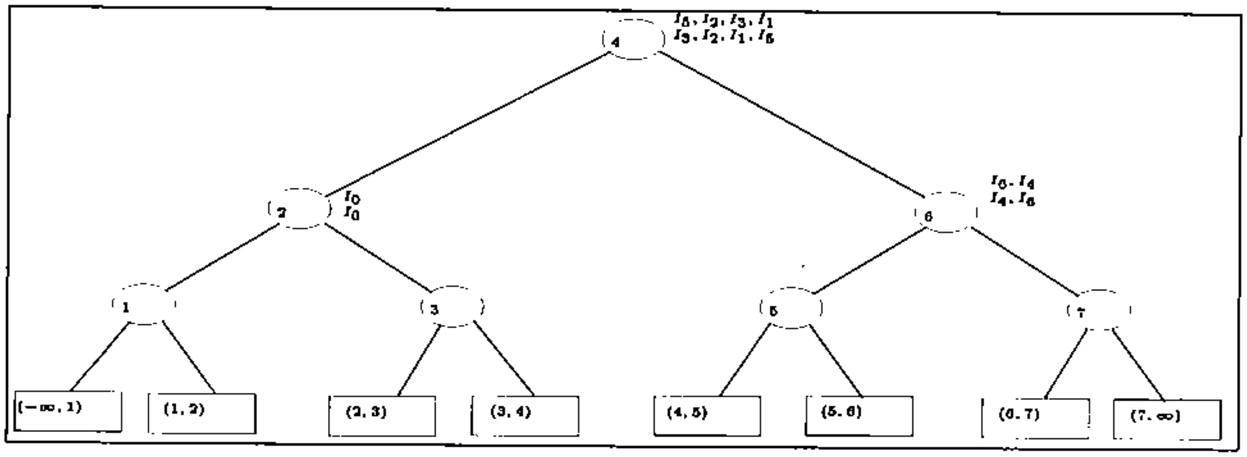


Figure 24: Interval tree for the intervals given in Figure 23

The total storage requirements can be broken down into *per interval* and *per node* costs. For each interval (seed cell), there are two float values for the extrema of the interval, one integer to store the cell identifier, and two indices in the sorted lists, for a total of $5n_s$ words of storage. Each node in the tree contains a split value, an integer number of cells stored at the node, and two pointers to the sorted lists of intervals, for a total of $4n_u$ words of storage. The total storage requirement is $5n_s + 4n_u$.

3.1.2 Interval Tree Construction

An interval tree is constructed in three steps. First, a sorted list of unique extreme values of intervals is created. This list forms an implicit binary tree, with the root node for the list $[i_a, i_b]$ taken as $i_{mid} = \lfloor (i_a + i_b)/2 \rfloor$ with left child $[i_a, i_{mid} - 1]$ and right child $[i_{mid} + 1, i_b]$. The second step for interval tree construction requires the iterative insertion of each interval into the tree. Finally, the *min-list* and *max-list* associated with each node is sorted as described above. The overall algorithm can be described as follows:

```
InsertInterval(tree, left, right, interval)
    $mid = \lfloor (left + right)/2 \rfloor$
    if tree.intervals[interval].max < tree.nodes[mid].split_value
    then
        InsertInterval(tree, left, mid-1, interval)
    else if tree.intervals[interval].min > tree.nodes[mid].split_value
    then
        InsertInterval(tree, mid+1, right, interval)
    else
        add interval to tree.nodes[mid]
    endif
BuildIntervalTree(tree)
    sort list of interval values
    store unique sorted list in split_value
    for each interval i
    do
        InsertInterval(tree, 0, tree.n_uniq_val-1, i)
    done
    for each node n
    do
        sort tree.nodes[n].min_list by increasing maximum value
        sort tree.nodes[n].max_list by decreasing minimum value
    done
```

The overall cost of building the interval tree is $O(n_s \log n_s)$, dominated by the initial cost of sorting the interval values.

3.1.3 Interval Tree Queries

For a given query value w, the reporting of intersected intervals is performed by a modified binary search for w:

```
QueryIntervalTree(tree, w)
    left = 0
    right = tree.n_uniq_val-1
    while left < right
    do
        mid = $\lfloor (left + right)/2 \rfloor$
        if $w >$ tree.nodes[mid].split_value
        then
            traverse tree.nodes[mid].min_list reporting intervals with $min < w$
            right = mid-1
        else
            traverse tree.nodes[mid].max_list reporting intervals with $max > w$
            left = mid+1
        endif
    done
```

The total cost for resolving the query is $O(k + \log n_u)$, where k is the size of the output and n_u is the number of unique values from the set of *min/max* values.

3.2 Segment Tree

A segment tree also consists of a binary search tree over the set of *min* and *max* values of all the seed cells [20, 22]. The primary difference from the interval tree is the manner in which the segments are stored. Nodes in a segment tree form a multiresolution hierarchy of intervals, with the root representing the infinite line, and with each node dividing the parent interval at a split value (see Figure 25). When a segment is inserted into the tree, it is recursively split and propagated downward in the tree to be inserted into the group of nodes whose intervals collectively sum to the entire range of the segment. Each segment will be stored at most $O(\log n_u)$ times, where $\log n_u$ is the height of the tree, resulting in worst case storage complexity of $O(n_s \log n_u)$ in the improbable case that all *min-max* values are distinct, and all intervals filter all the way down to the leaves. The query complexity for reporting the k intersected cells for a given isovalue w is $O(k + \log n_u)$.

3.2.1 Segment Tree Data Structure

The segment tree data structure is similar to that of the interval tree. Note that for the segment tree there is no need to explicitly store the *min/max* values for each segment. As illustrated in Figure 25, there are three principal lists of cells associated with each unique interval value. We group these three lists into one segment tree *node*, as shown below. The tree structure is again implicit in the sorted ordering of the unique values.

```
struct ST_CellList {
   int n_cells;
   int *cell_id;
};

struct ST_Node {
   float       split_value;
   ST_CellList lth_list;
   ST_CellList leq_list;
   ST_CellList geq_list;
};
```

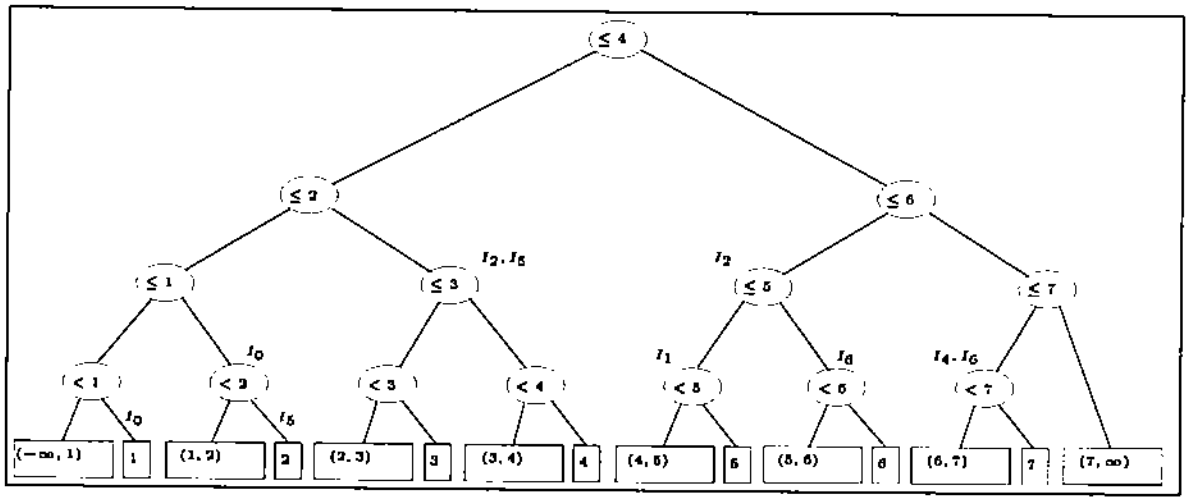


Figure 25: Segment tree for the segments given in Figure 23

```
struct SegmentTree {
   int          n_intervals;

   int          n_uniq_val;
   ST_Node     *nodes;
};
```

In the case of the segment tree, total storage is dependent on the number of times each interval is split. We will introduce n_{i_s} as the total number of cell identifiers stored in all lists of cells. For each node of the tree, we have the float split value, three pointers to cells and three counters for the number of cells in each list, for a total of $n_{i_s} + 6n_u$ words of storage overall.

3.2.2 Segment Tree Construction

Construction of a segment tree is very similar to construction of an interval tree. The same binary structure is constructed over the unique extreme values of the seed cells. The primary difference is that each interval is recursively split and propagated down the tree from the root, rather than terminating at the first "split-value" which is spanned by the interval. The algorithm is sketched below:

```
InsertSegment(tree, left, right, cell_id, min, max, imin, imax)
    $mid = \lfloor (left + right)/2 \rfloor$
    $split\_value = tree.nodes[mid].split\_value$
    if $left = right$
    then
        if $min < imax$
        then
            add cell to $tree.nodes[mid].lth\_list$
        else
            add cell to $tree.nodes[mid].geq\_list$
        endif
        return
    endif

    if $min \leq imin$ AND $max \geq imax$
    then
        add cell to $tree.nodes[mid].leq\_list$
        return
        endif

        if $min <$ tree.nodes[mid].split_value
        then
            InsertSegment(tree, left, mid, cell_id, min, MIN(max, split_value),
            split_value, imax)
        endif

        if $max >$ tree.nodes[mid].split_value
        then
            InsertSegment(tree, mid+1, right, cell_id, MAX(min, split_value), max,
            imin, split_value)
        endif
    BuildSegmentTree(tree)
        sort list of interval values
        store unique sorted list in split_value
        for each interval $i$
        do
            InsertSegment(tree, 0, tree.n_uniq_val-1, cell_id, min, max, $-\infty$, $\infty$)
        done
```

3.2.3 Segment Tree Queries

Traversal of a segment tree is much like traversal of an interval tree. For a given query value w, the reporting of intersected intervals is performed by a modified binary search for w. As each node is traversed, the associated list of cells is selected. At the conclusion of the traversal, one or both of the remaining two lists is selected, as outlined below:

```
    QuerySegmentTree(tree, w)
        left = 0
        right = tree.n_uniq_val-1
        while left < right
        do
            mid = $\lfloor (left + right)/2 \rfloor$
            traverse tree.nodes[mid].leq_list and report all cells
            if $w \leq$ tree.nodes[mid].split_value
            then
                right = mid
            else
                left = mid+1
            endif
        traverse tree.nodes[left].lth_list and report all cells
        if $w =$ tree.nodes[left].split_value
        then
            traverse tree.nodes[left].geq_list and report all cells
        done
```

The total cost for resolving the query is $O(k + \log n_u)$, where k is the size of the output and n_u is the number of unique values from the set of *min/max* values.

3.3 Bucket Search

Much of the scientific data that we are concerned with comes in the form of integer values in a small range. For example, Computed Tomography (CT) data generally have a 12-bit integer range of values. This regular subdivision allows a simple bucket search strategy with $n_u - 1$ buckets each representing a unit interval $(h, h+1)$. For each cell, an identifier is stored in each bucket which is spanned by the cell. Clearly, the worst case storage complexity of this strategy is $O(n_s n_u)$, which may be infeasible in the case in which all cells are stored. Given the approach of forming a small set of seed cells, such a technique may prove feasible, with the added benefit of allowing intersected cells to be reported in $O(k)$ time, linear in the number of reported cells.

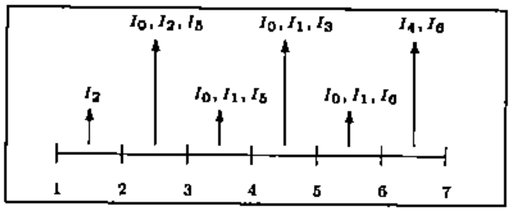


Figure 26: Bucket search structure for the intervals given in Figure 23

3.3.1 Data Structure

```
struct B_CellList {
   int n_cells;
   int *cell_list;
};

struct BucketSearch {
   int           min, max;
   B_CellList   *lists;
};
```

As in the case of a segment tree, each cell may be stored several times, and so we will use n_{i_b} to represented the total number of cell identifiers stored. In addition, we have one list for each unique extreme value, and so the total measured storage is $n_{i_b} + 2n_u$.

3.3.2 Building a Bucket Structure

The creation of a bucket data structure is straightforward. For each bucket spanned by a cell, it is added to the associated list.

```
InsertInBuckets(bucket, cell_id, min, max)
    for $b$ = min to max−1
    do
        add cell to bucket.lists[b].cell_list
    done
BuildBucketSearch(bucket)
    sort list of interval values
    store unique sorted list in split_value
    for each interval $i$
    do
        InsertInBuckets(tree, cell_id, min, max)
    done
```

The time required for building the search structure is proportional to the total number of buckets spanned by all cells, in worst case $O(n_s n_u)$.

3.3.3 Bucket Search Queries

The advantage of the bucket search structure is that the range query complexity is entirely output sensitive, $O(k)$. The procedure is outlined below:

```
QueryBucket(search, w)
    bucket = w - bucket.min
    traverse search.lists[bucket].cell_list and report all cells
```

3.4 Search Structure Discussion

In this section we discuss the storage cost of each of the three presented search structures. Table 2 summarizes the theoretical space and query complexities.

Search Structure	Storage Complexity	Query Complexity
Interval Tree	$O(n_s)$	$O(k + \log n_u)$
Segment Tree	$O(n_s \log n_u)$	$O(k + \log n_u)$
Bucket	$O(n_s n_u)$	$O(k)$

Table 2: Comparison of the theoretical complexities of the three search structures for performing an interval query.

In examining the practical considerations, we have measured the storage of each data structure as shown in Table 3.

Search Structure	Storage Measure
Interval Tree	$5n_s + 4n_u$
Segment Tree	$n_{i_s} + 6n_u$
Bucket	$n_{i_b} + 2n_u$

Table 3: Comparison of the storage requirements in typical implementation of the three search structures.

It is clear from both the theoretical complexities and the empirical storage measures that the actual search structure size will depend on certain characteristics of the data. In particular, if n_u bounded (such as in the case of integer data), the theoretical storage and query complexities are the same for all three search structures.

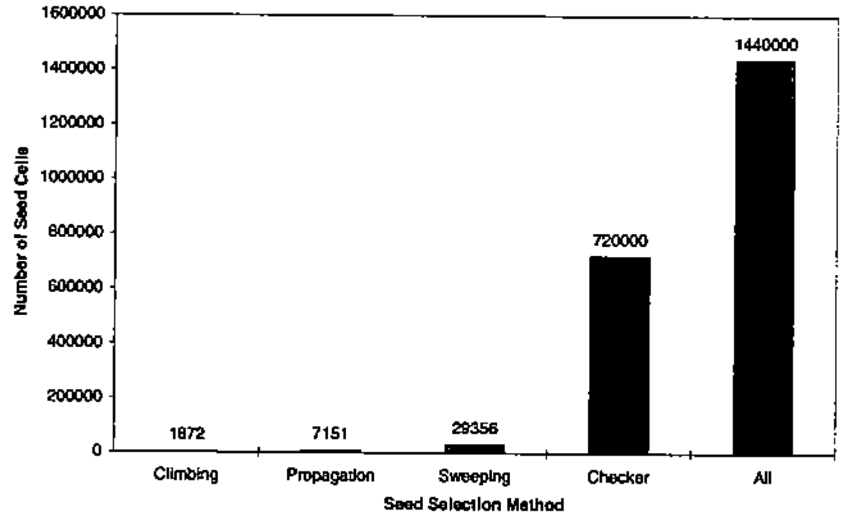
Empirical results from the three seed set construction algorithms are given in Figures 27-31. We also compare the results with the size of an interval tree for the the checkerboard seed set as well as the entire set of cells. Note that for data with integer values (the terrain data and the SOD data), the size of the segment tree is smaller than that of the interval tree, contrary to what the theoretical complexities might lead one to expect. In Figures 32-34 we compare the total preprocessing times for each seed selection algorithm. All times in these graphs are computed using the interval tree as a search structure. Note, in particular, that the total preprocessing time for the directional sweep is actually less than using the checkerboard approach or using the entire set of cells, simply because the directional sweep has time complexity $O(n_c)$ and the construction of the interval tree is $O(n_s \log n_u)$. The directional sweep is extremely fast and reduces size of the seed set sufficiently to actually provide an observed time cost advantage over all other approaches tested. Figure 35 displays an average cost of performing isovalue queries for an MRI dataset of size 256x256. Note that due to the fast inner loop of the segment tree and bucket search structure query algorithms, both exhibit an advantage in query time over the interval tree.

References

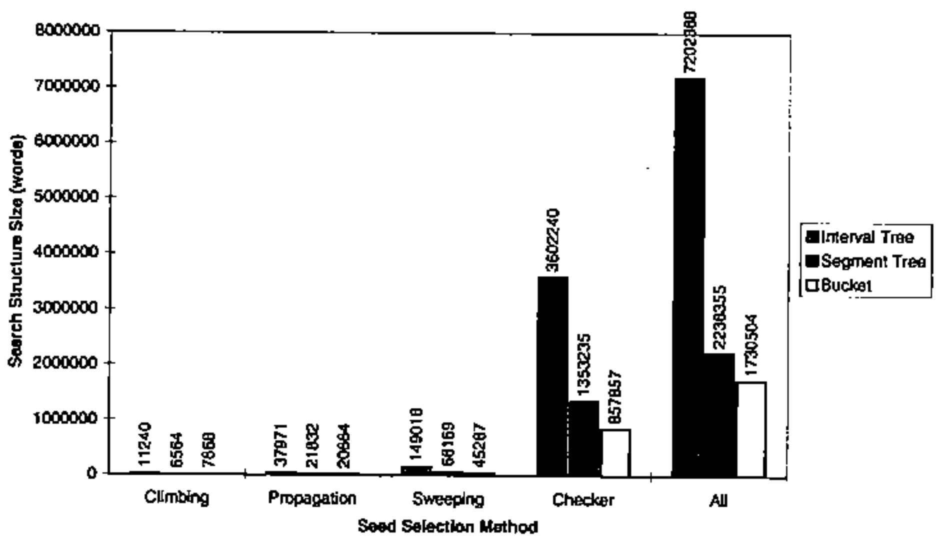
[1] E. Artzy, G. Frieder, and G. T. Herman. The theory, design, implementation and evaluation of a three-dimensional surface detection algorithm. In *Computer Graphics (SIGGRAPH '80 Proceedings)*, volume 14, pages 2–9, 1980.

[2] C. L. Bajaj, V. Pascucci, and D. R. Schikore. Fast isocontouring for improved interactivity. In *Proceedings of 1996 Symposium on Volume Visualization*, pages 39–46, October 1996.

[3] P. Cignoni, C. Montani, E. Puppo, and R. Scopigno. Optimal isosurface extraction from irregular volume data. In *Proceedings of 1996 Symposium on Volume Visualization*, pages 31–38, October 1996.

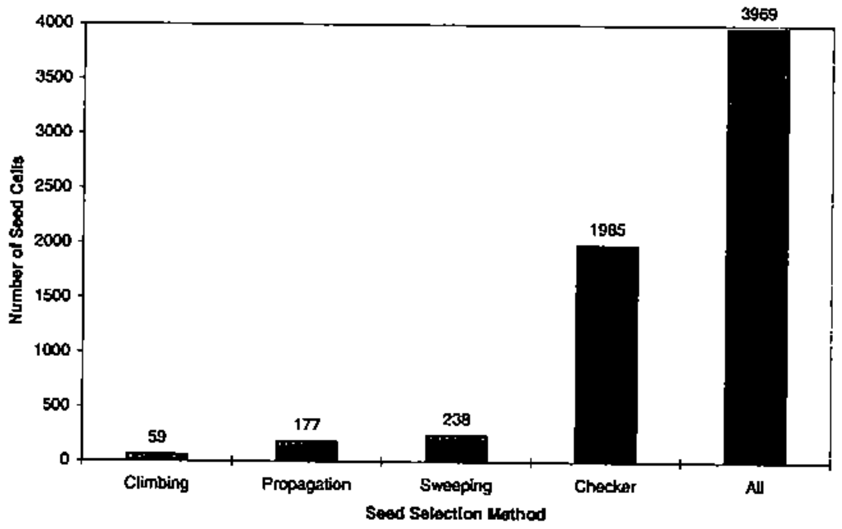


(a)

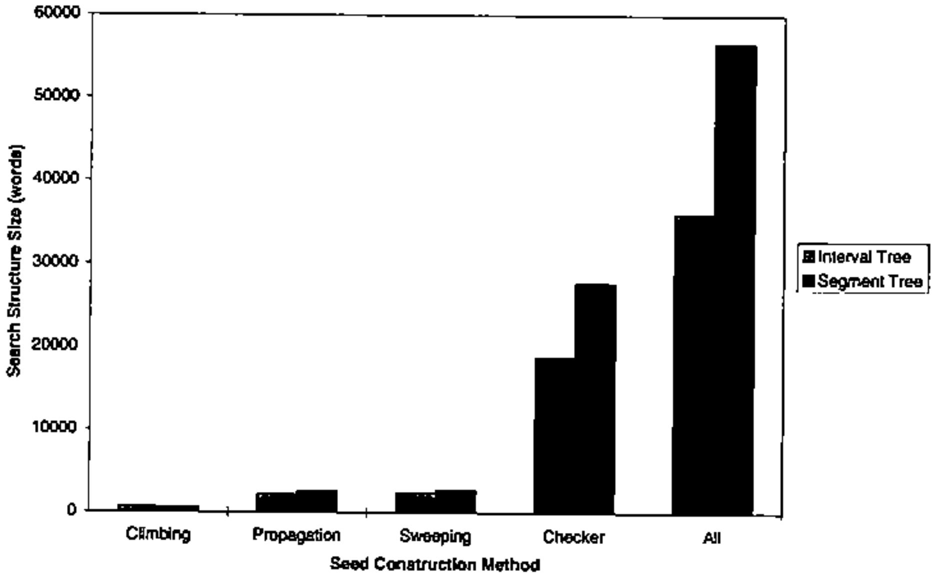


(b)

Figure 27: Number of seeds (a) and search structure storage requirements (b) for the Eagle Pass USGS Terrain Data (1201x1201)

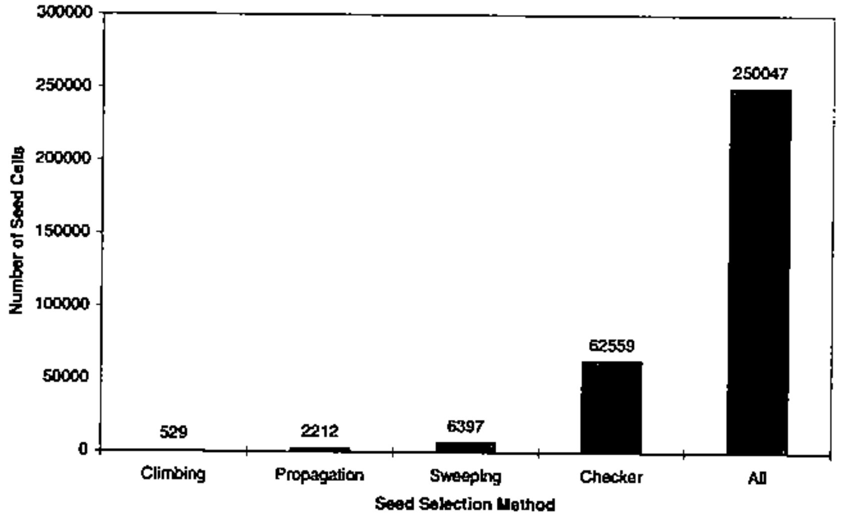


(a)

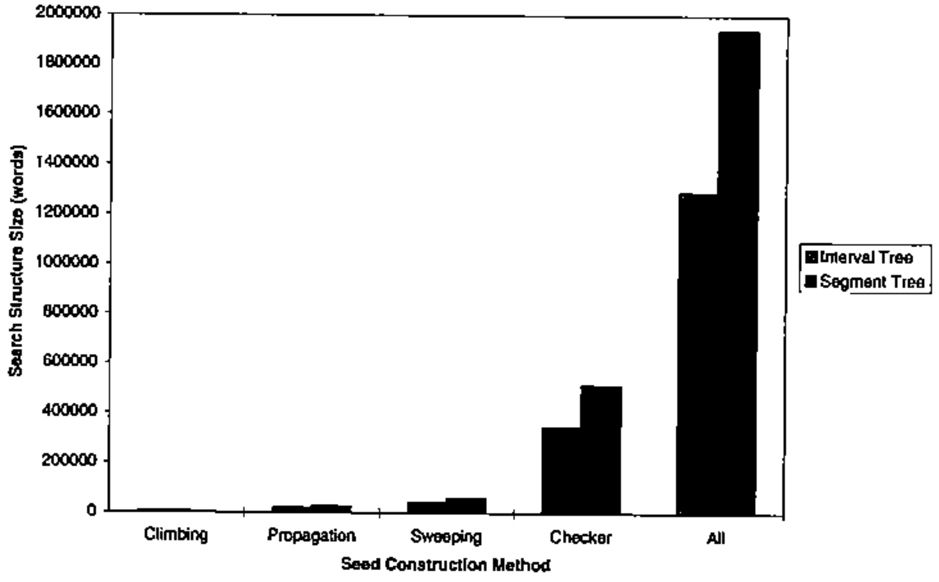


(b)

Figure 28: Number of seeds (a) and search structure storage requirements (b) for a sample function (64x64)

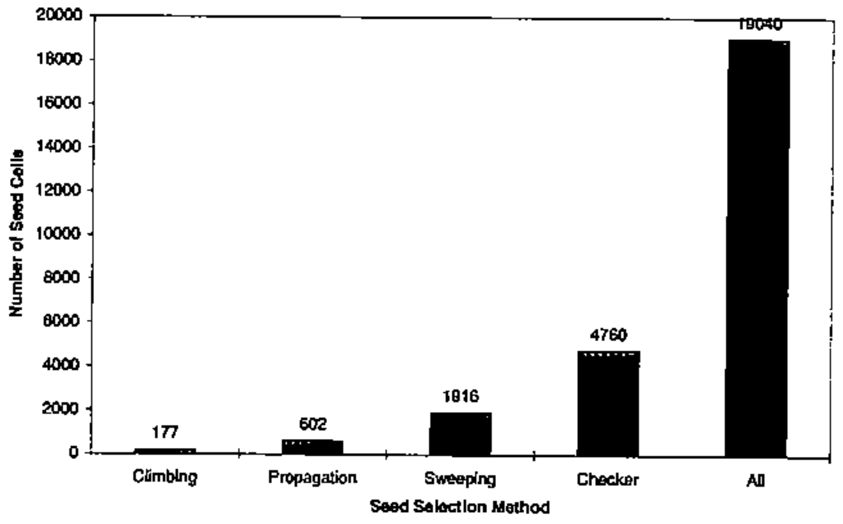


(a)

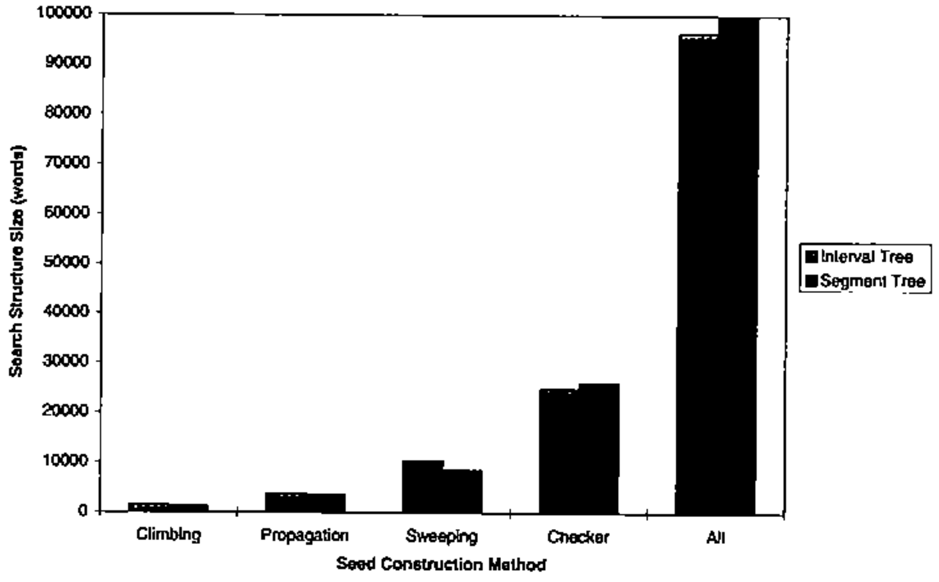


(b)

Figure 29: Number of seeds (a) and search structure storage requirements (b) for the Hipip data (64x64x64)

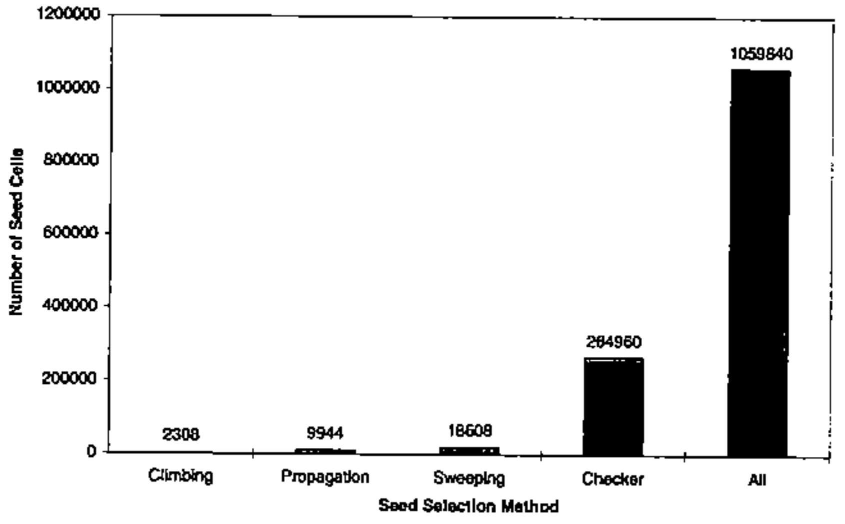


(a)

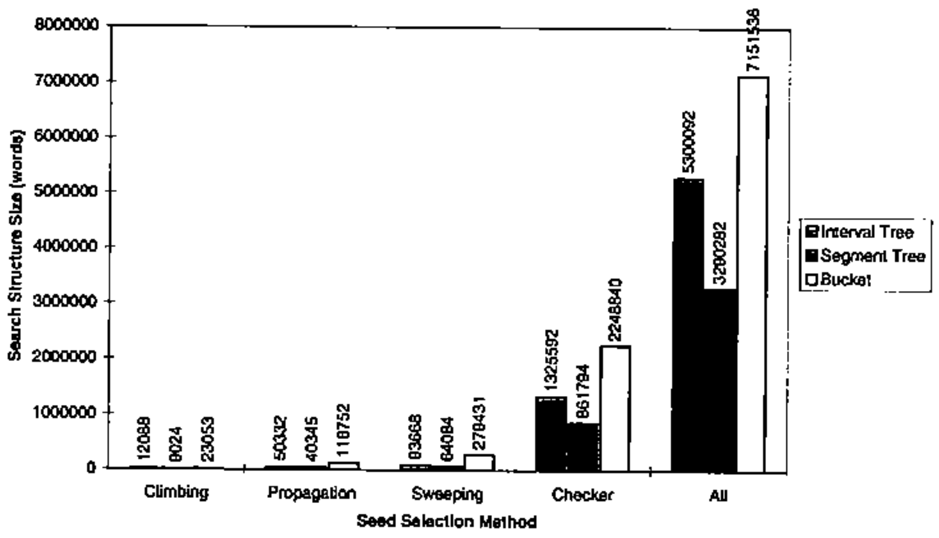


(b)

Figure 30: Number of seeds (a) and search structure storage requirements (b) for the LAMP Climate Data (35x41x15)

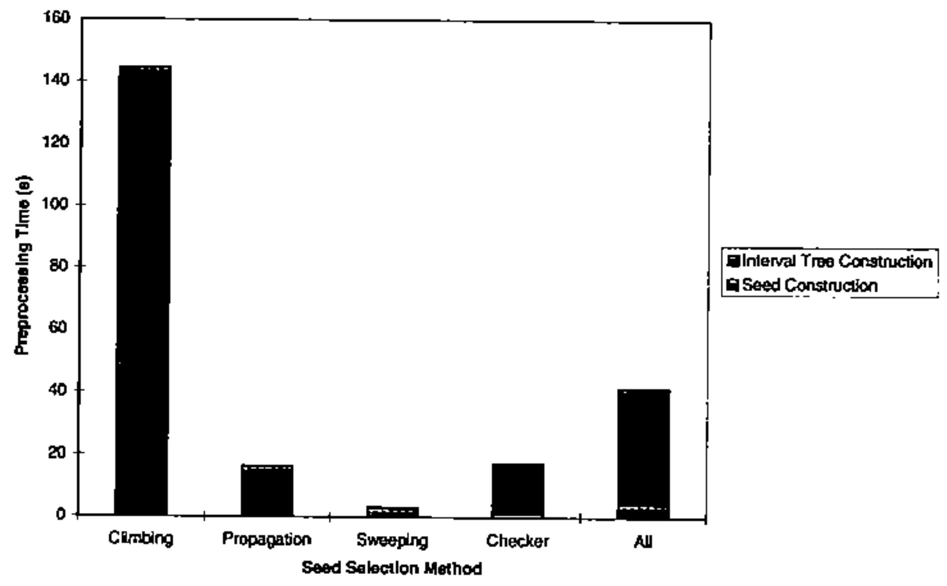


(a)

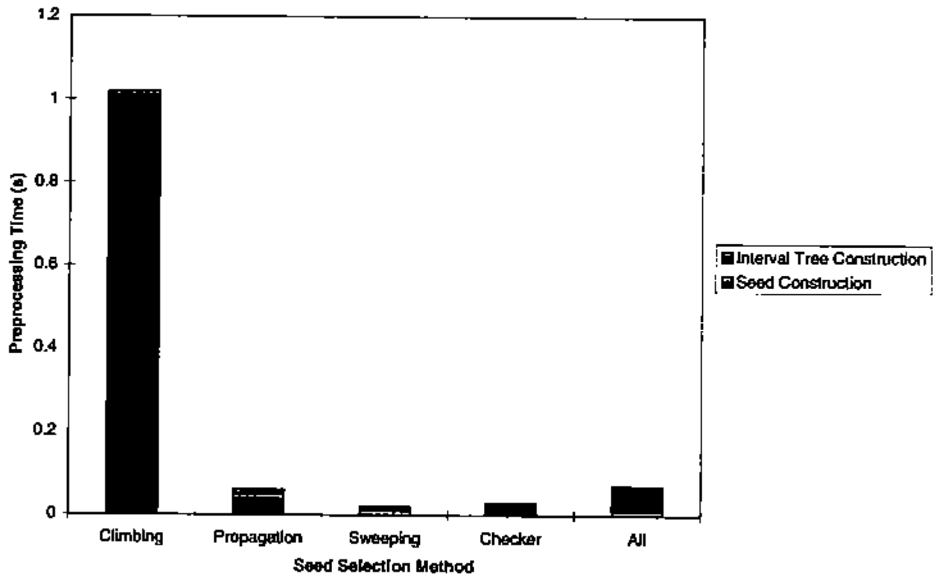


(b)

Figure 31: Number of seeds (a) and search structure storage requirements (b) for the SOD data (97x97x116)

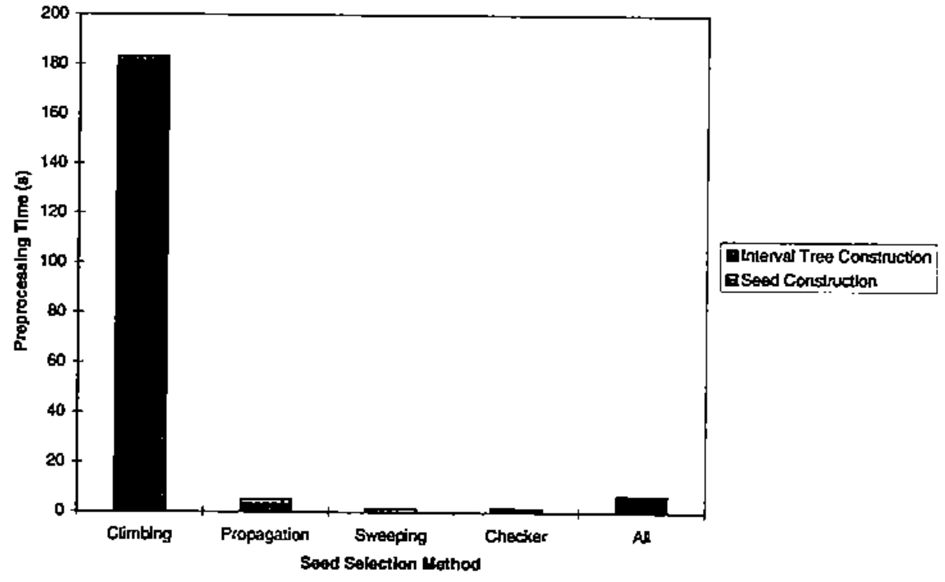


(a) Eagle Pass Data (1201x1201)

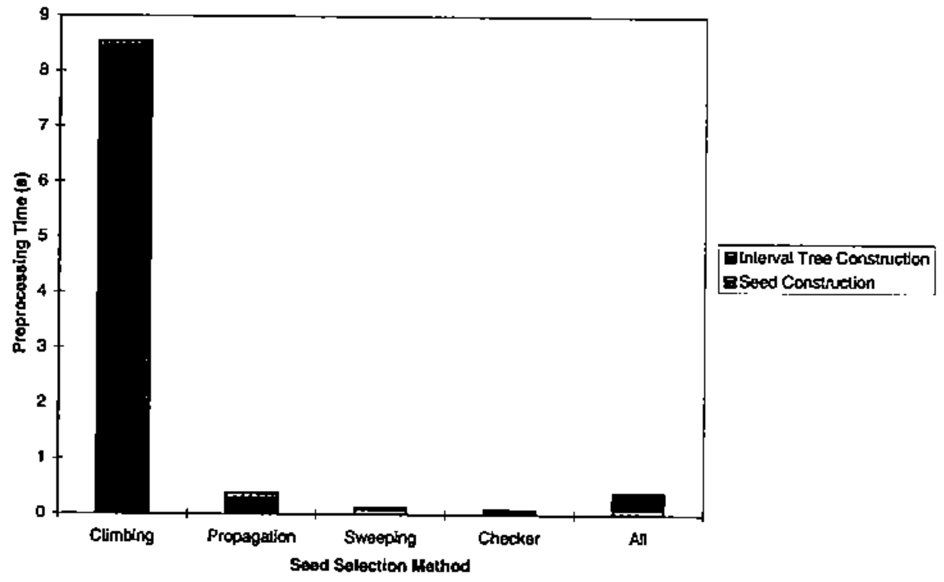


(b) Sample Function (64x64)

Figure 32: Comparison of preprocessing time required for the 5 seed cell extraction algorithms. The interval tree is used as the search structure in all cases.

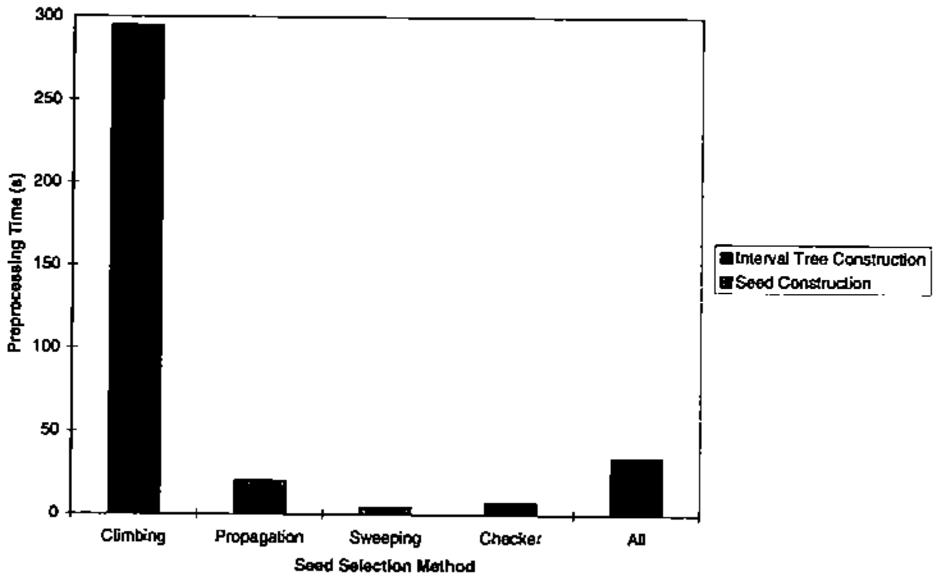


(a) Hipip Data (64x64x64)



(b) LAMP Climate Model (35x41x15)

Figure 33: Comparison of preprocessing time required for the 5 seed cell extraction algorithms. The interval tree is used as the search structure in all cases.



(a)

Figure 34: Comparison of preprocessing time required for the 5 seed cell extraction algorithms. The interval tree is used as the search structure in all cases.

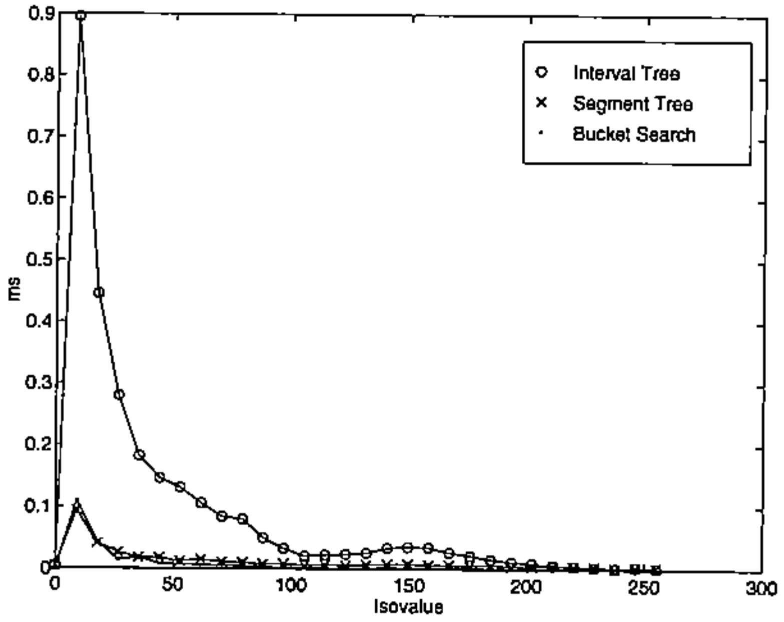


Figure 35: Comparison of query time for the interval tree, the segment tree, and the bucket search structures. Query time computed as an average over 1000 searches and is plotted as a function of isovalue.

[4] P. Cignoni, C. Montani, E. Puppo, and R. Scopigno. Speeding up isosurface extraction using interval trees. *IEEE Transactions on Visualization and Computer Graphics*, 3(2):158–170, 1997.

[5] A. Doi and A. Koide. An efficient method of triangulating equi-valued surfaces by using tetrahedral cells. *IEICE Trans. Commun. Elec. Inf. Syst.*, E-74(1):214–224, 1991.

[6] M. J. Durst. Additional reference to marching cubes. *Computer Graphics*, 22(2):72–73, 1988.

[7] R. S. Gallagher. Span filtering: An efficient scheme for volume visualization of large finite element models. In G. M. Nielson and L. Rosenblum, editors, *Proceedings of IEEE Visualization '91*, pages 68–75, October 1991.

[8] R. Haimes. Techniques for interactive and interrogative scientific volumetric visualization. Available from `http://raphael.mit.edu/ visual3/visual3.html`, October 1991.

[9] C. T. Howie and E. H. Blake. The mesh propagation algorithm for isosurface construction. *Computer Graphics Forum*, 13(3):65–74, 1994. Eurographics '94 Conference issue.

[10] T. Itoh and K. Koyamada. Automatic isosurface propagation using an extrema graph and sorted boundary cell lists. *IEEE Transactions on Visualization and Computer Graphics*, 1(4):319–327, December 1995.

[11] T. Itoh, Y. Yamaguchi, and K. Koyamada. Volume thinning for automatic isosurface propagation. In *Proceedings of IEEE Visualization '96*, pages 303–310, October 1996.

[12] D. B. Karron, J. Cox, and B. Mishra. New findings from the spiderweb algorithm: Toward a digital Morse theory. In *Visualization in Biomedical Computing*, volume 2359, pages 643–657. SPIE, October 1994.

[13] D. Kenwright. *Dual Stream Function Methods for Generating Three-Dimensional Streamlines.* Ph.D. thesis, University of Auckland, Australia, 1993.

[14] Y. Livnat, H.-W. Shen, and C. R. Johnson. A near optimal isosurface extraction algorithm for unstructured grids. *IEEE Transactions on Visualization and Computer Graphics*, 2(1):73–84, 1996.

[15] S. Lobregt, P. W. Verbeek, and F. C. A. Groen. Three-dimensional skeletonization: Principle and algorithms. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2(1):75–77, 1980.

[16] W. E. Lorensen. Marching through the visible man. In G. M. Nielson and D. Silver, editors, *Proceedings of IEEE Visualization '95*, pages 368–373, October 1995.

[17] W. E. Lorensen and H. E. Cline. Marching cubes: A high resolution 3D surface construction algorithm. In Maureen C. Stone, editor, *Computer Graphics (SIGGRAPH '87 Proceedings)*, volume 21, pages 163–169, July 1987.

[18] S. V. Matveyev. Approximating of isosurface in the marching cube: Ambiguity problem. In R. D. Bergeron and A. E. Kaufman, editors, *Proceedings of Visualization '94*, pages 288–292. IEEE Computer Society, IEEE Computer Society Press, October 1994.

[19] E. M. McCreight. Priority search trees. *SIAM J. Comput.*, 14:257–276, 1985.

[20] K. Mehlhorn. *Data Structures and Algorithms 3: Multi-dimensional Searching and Computational Geometry*, volume 3 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, Heidelberg, West Germany, 1984.

[21] C. Montani, R. Scateni, and R. Scopigno. Discretized marching cubes. In R. D. Bergeron and A. E. Kaufman, editors, *Proceedings of IEEE Visualization '94*, pages 281–287. IEEE Computer Society, IEEE Computer Society Press, October 1994.

[22] K. Mulmuley. *Computational Geometry: An Introduction Through Randomized Algorithms.* Prentice Hall, Englewood Cliffs, NJ, 1994.

[23] B. K. Natarajan. On generating topologically consistent isosurfaces from uniform samples. *The Visual Computer*, 11(1):52–62, 1994.

[24] G. M. Nielson and B. Hamann. The asymptotic decider: Resolving the ambiguity of marching cubes. In G. M. Nielson and L. Rosenblum, editors, *Visualization '91 Proceedings*, pages 83–91, October 1991.

[25] H.-W. Shen, C. D. Hansen, Y. Livnat, and C. R. Johnson. Isosurfacing in span space with utmost efficiency. In *Visualization '96 Proceedings*, pages 287–294, October 1996.

[26] H.-W. Shen and C. R. Johnson. Sweeping simplices: A fast iso-surface extraction algorithm for unstructured grids. In G. M. Nielson and D. Silver, editors, *Proceedings of IEEE Visualization '95*, pages 143–150, October 1995.

[27] S. N. Srihari. Representation of three-dimensional digital images. *Computing Surveys*, 13(4):399–424, 1981.

[28] A. Van Gelder and J. Wilhelms. Topological considerations in isosurface generation. Technical Report UCSC-CRL-94-31, University of California at Santa Cruz, June 1994.

[29] A. Van Gelder and J. Wilhelms. Topological considerations in isosurface generation. *ACM Transactions on Graphics*, 13(4):337–375, October 1994.

[30] M. van Kreveld. Efficient methods for isoline extraction from a digital elevation model based on triangulated irregular networks. *International Journal of Geographical Information Systems*, 10:523–540, 1996. Also appeared as Technical Report UU-CS-1994-21, University of Utrecht, the Netherlands.

[31] M. van Kreveld, R. van Oostrum, C. L. Bajaj, V. Pascucci, and D. R. Schikore. Contour trees and small seed sets for isosurface traversal. In *13th ACM Symposium on Computational Geometry*, pages 212–220. ACM, 1997.

[32] J. J. van Wijk. Implicit stream surfaces. In *Proceedings of IEEE Visualization '93*, pages 245–252, October 1993.

[33] J. Wilhelms and A. Van Gelder. Octrees for faster isosurface generation extended abstract. *Computer Graphics (San Diego Workshop on Volume Visualization)*, 24(5):57–62, November 1990.

[34] J. Wilhelms and A. Van Gelder. Topological considerations in isosurface generation: Extended abstract. *Computer Graphics (San Diego Workshop on Volume Visualization)*, 24(5):79–86, November 1990.

[35] J. Wilhelms and A. Van Gelder. Octrees for faster isosurface generation. *ACM Transactions on Graphics*, 11(3):201–227, 1992.

[36] G. Wyvill, C. McPheeters, and B. Wyvill. Data structure for soft objects. *The Visual Computer*, 2:227–234, 1986.

[37] Y. Zhou, W. Chen, and Z. Tang. An elaborate ambiguity detection method for constructing isosurfaces within tetrahedral meshes. *Computers and Graphics*, 19(3):355–364, 1995.