On-Demand Augmentation of Contour Trees

Mohit Sharma
Indian Institute of Science, Bangalore
mohitsharma@iisc.ac.in

Vijay Natarajan
Indian Institute of Science, Bangalore
vijayn@iisc.ac.in

ABSTRACT

The contour tree represents the topology of level sets of a scalar function. Nodes of the tree correspond to critical level sets and arcs of the tree represent a collection of topologically equivalent level sets connecting two critical level sets. The augmented contour tree contains degree-2 nodes on the arcs that represent regular level sets. The degree-2 nodes correspond to regular points of the scalar function and other critical points that do not affect the number of level set components. The augmented contour tree is significantly larger in size and requires more effort to compute when compared to the contour tree. Applications of the contour tree to data exploration and visualization require the augmented contour tree. Current approaches propose algorithms to compute the contour tree and the augmented contour tree from scratch. Precomputing and storing the large augmented contour tree will not be necessary if the contour tree can be augmented on-demand. This paper poses the problem of computing the augmented contour tree given a contour tree as input. Computational experiments demonstrate that the on-demand augmentation can be computed fast while resulting in good memory savings.

CCS CONCEPTS

• **Human-centered computing** → **Visualization techniques**; **Scientific visualization**; • **Theory of computation** → *Computational geometry*.

ACM Reference Format:
Mohit Sharma and Vijay Natarajan. 2018. On-Demand Augmentation of Contour Trees. In *11th Indian Conference on Computer Vision, Graphics and Image Processing (ICVGIP 2018), December 18–22, 2018, Hyderabad, India,* Anoop M. Namboodiri, Vineeth Balasubramanian, Amit Roy-Chowdhury, and Guido Gerig (Eds.). ACM, New York, NY, USA, Article 104, 8 pages. https://doi.org/10.1145/3293353.3293384

1 INTRODUCTION

Data from science and engineering disciplines is often represented as a scalar function over a geometric domain. For example, medical imaging data such as from CT or MRI scans is available as a stack of 2D images and represented as a 3D scalar function. Temperature, pressure, and precipitation data from weather and climate simulations or satellite imagery is represented as a 2D scalar function over the surface of earth or a 3D scalar function. Rapidly increasing computational power has resulted in the generation of high resolution data that are large in size and rich in number of features. This increase has necessitated the development of new feature-aware techniques for visualizing and exploring the data. Topology-based methods provide an abstract representation of data and have been successfully used to represent, store, query, and explore these rich data sets.

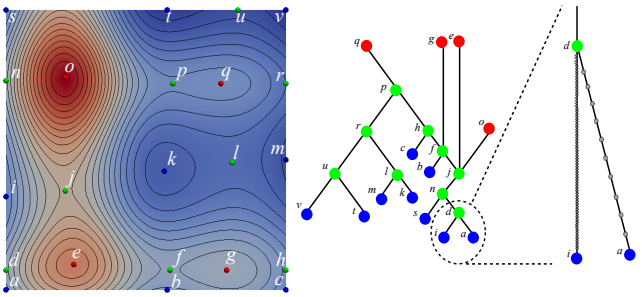


Figure 1: Contour tree for a 2D scalar function. A level set is the preimage of a real value. Each connected component of a level set is mapped to a point in the contour tree. Maxima are shown in red, minima in blue, and merge/split saddles in green. (Zoomed in) Augmented contour tree contains the degree-2 nodes also, shown in gray, which correspond to remaining vertices of the input domain.

The contour tree is a popular and well studied topological structures in the visualization and computational geometry community [2, 3, 25, 26]. A *level set* of a scalar function is a preimage of a real value. The contour tree tracks the evolution of the connectivity of level sets of a scalar function. Figure 1 shows the contour tree for a 2D scalar function. Note that the nodes of the tree correspond to critical points of the scalar function, namely minima, maxima, and saddles. The nodes have degree 1 or 3. The *augmented contour tree* also contains degree-2 nodes corresponding to the regular points of the scalar function and other critical points that do not affect the number of level set components. The augmented contour tree is useful to compute a mapping from the tree into the domain of the scalar function. However, precomputing and storing the augmented contour tree is costly because its size is comparable to the size of the input. In comparison, the contour tree is much smaller in size in practice. Several applications require the mapping from arcs of the contour tree to sub-domain regions [8, 21, 25]. We address this requirement by describing a simple algorithm for augmenting an existing contour tree on demand. We describe results of computational experiments on different datasets ranging from a few thousand to a million vertices to demonstrate the storage benefits and efficiency of the method in practice.

ICVGIP 2018, December 18–22, 2018, Hyderabad, India

ACM ISBN 978-1-4503-6615-1/18/12…$15.00
https://doi.org/10.1145/3293353.3293384

1.1 Related work

The contour tree was first studied in the context of a GIS application by de Berg and van Kreveld [6]. They proposed a divide and conquer approach to compute the contour tree of the elevation function defined over a 2D domain. This algorithm computes the contour tree in $O(n \log n)$ time, where n represents the number of triangles in the mesh representing the domain of the elevation function. Kreveld et al. [24] described an algorithm that efficiently tracks level set components to compute the contour tree in $O(n \log n)$ time for 2D scalar functions and in $O(n^2)$ for 3D scalar functions. Tarasov and Vyalyi [20] improved upon earlier methods by performing two sweeps over the domain, one in increasing and another in decreasing function value to identify the merges and splits in the level set components. In a final step, they merge the results of the two sweeps to compute the contour tree in $O(n \log n)$ time for 3D input. Carr et al. [2] simplify the sweep-based approach even further to compute a join and split tree from two sweeps. The two trees are merged to construct the contour tree. The algorithm computes the contour tree for scalar functions defined over any d-dimensional domain and runs in $O(v \log v + n\alpha(n))$, where n is the number of tetrahedra and v is the number of vertices in the input domain, and α is the inverse Ackermann function.

Chiang et al. [4] proposed a different approach, one that traces monotone paths to directly identify arcs in the contour tree. The method first queries the local neighborhood of all vertices to identify component critical points. Next, it traces monotone paths between these critical points to construct the join and split trees. The algorithm takes $O(t \log t + n)$ time, where t is the number of critical points and n is the number of tetrahedra in domain. This algorithm is provably optimal based on a $\Omega(t \log t)$ lower bound for contour tree construction [23]. The contour tree is the loop free version of the *Reeb graph* [18], which has also been extensively studied [5, 9, 17, 22].

Parallel algorithms have also been developed for computing the contour tree. These algorithms are primarily data parallel and word on data sampled on a grid, however a few task parallel methods have also been developed. Pascucci and Cole-McLaughlin [16] described a data parallel algorithm that works on data sampled at vertices of a 3D grid and computes the contour tree for the piecewise polynomial function obtained using a trilinear interpolant. The sequential version runs in $O(n + t \log n)$ time, where t is the number of critical points and n is the number of vertices. The parallel algorithm provides a linear speed up with the number of processors.

Acharya and Natarajan [1] developed an algorithm that uses a hybrid approach resulting in good improvements in terms of running time and memory usage. The grid domain is split into smaller sub-domains using an octree based subdivision. Local join and split trees are computed for these sub-domains using a monotone path tracing based method. The local trees are stitched together to construct global join and split trees, which are in turn merged together in a final step resulting in the contour tree. The sequential running time for this method is $O(n + (t + n^{2/3})log(t + n^{2/3}))$, where t is the number of critical points and n is the total number of vertices.

More recently, Gueunet et al. [11] presented a task parallel algorithm to compute augmented merge trees. The worst case running time for this algorithm is equal to the traditional sequential algorithms but it is very fast in practice. The algorithm uses a clever sequence of breadth first search traversals and a priority queue to identify individual arcs of the augmented merge tree. Other parallel algorithms, distributed [12, 14, 15] and shared memory [13], have also been proposed. However these algorithms do not typically compute the augmented version of the contour tree. An exception is the work of Gueunet et al., which considers the augmented version but focuses on the relatively simpler merge trees.

1.2 Contributions

In this paper, we pose the problem of augmenting an existing contour tree. Several applications of the contour tree require a mapping between selected arcs of the contour tree and the corresponding subset of the domain. We present a simple and efficient algorithm that processes an input contour tree together with the domain of the scalar function to produce the contour tree, whose arcs are augmented with all degree-2 nodes. The algorithm uses a breadth first search traversal similar to Gueunet et al. [11] to identify a degree-2 nodes corresponding to an arc but avoids the use of a priority queue thereby saving computation time. The algorithm can be easily incorporated into existing visualizations and analysis workflows.

The augmented contour tree is typically much larger in size when compared to the contour tree. We demonstrate via computational experiments that the savings in terms of memory usage is indeed significantly large, while the time required to augment the tree is reasonable. The key benefit is that the augmentation algorithm may be invoked on demand and hence results in large saving (typically 10×–20×) in terms of storage. We also describe applications such as feature-aware selection and isocontour extraction that will benefit from the on-demand augmentation.

2 LEVEL SET TOPOLOGY

We begin by introducing the notion of level set topology and how it is represented within the contour tree. We refer the reader to a text on computational topology [10] for a more detailed description of the relevant terms and definitions.

A scalar function f is often represented as a sample over vertices of a mesh that represents the input domain. The samples are interpolated within the interior of the mesh elements. The mesh may be a structured grid such as a lattice grid or a triangulation. In the case of a lattice grid, the scalar values at the vertices are extended to the interior of the grid cells via trilinear interpolation. The term scalar value and function value are used interchangeably. A *level set* corresponding to a scalar value α is the preimage $f^{-1}(\alpha)$ of α. It is also referred as the isocontour or isosurface, the set of all points on the domain where the function value is equal to α. A *contour* is a connected component of a level set.

Figure 1 shows a 2D scalar function and a few of its level sets. Consider a sweep over the domain in increasing or decreasing order of function value. Individual contours expand, shrink, or the level set possibly changes topology. We are specifically interested in the changes to the number of connected components of the level set. The different possible changes in topology include creation of a new contour, destruction of an existing contour, merging of two

contours, or splitting of a contour. The level set topology changes only when it passes past a critical point of the scalar function. The corresponding scalar value is called a critical value. During a sweep in increasing order of function value, a new level set component is created at a minimum, a component is destroyed at a maximum, two contours merge into one or a single contour splits into two at a saddle.

The *contour tree* is obtained by mapping each contour to a point. Nodes of the contour tree correspond to the critical points where level set topology changes. Arcs correspond to a collections of contours that are topologically equivalent to each other. These contours pass through vertices, which may be included into the arc as degree-2 nodes thereby resulting in an *augmented contour tree*.

The contour tree in Figure 1 contains 9 minima (blue) and 4 maxima (red), the leaf nodes of the contour tree. The merge and split nodes appear as degree-3 nodes (green). The degree-2 nodes (gray) do not affect the level set topology. Vertices a and i are two minima. Contours that originate from them merge at saddle d as seen in the contour tree. The degree-2 nodes on the arc ad correspond to vertices in the domain through which contours that originate at vertex a passed before reaching vertex d. In this paper, our aim is to locate these degree-2 nodes efficiently and augment the arc appropriately.

3 AUGMENTING A CONTOUR TREE

We now describe an algorithm that computes all degree-2 nodes corresponding to each arc of the contour tree.

3.1 Algorithm

The algorithm takes as input a contour tree (with degree-1 and degree-3 nodes) and the triangulation that represents the domain of the scalar function. It performs a sequence of breadth first search traversals on the input mesh, see Algorithm 1. The arcs of the contour tree are processed iteratively, beginning with an arc that is incident on a leaf node. The BFS traversal originating at a vertex that corresponds to the leaf node determines all degree-2 nodes that augment the arc. After processing, the arc is removed from the tree potentially transforming an interior node into a leaf node.

Figure 2 shows a small triangulation with a scalar function defined on its vertices and the corresponding contour tree. We will use this example to understand the intuition that drives the design of the algorithm. Labels within the vertices of the mesh and nodes of the tree correspond to the function value. The algorithm reports the collection of gray vertices that belong to each arc of the contour tree.

In order to augment an arc (4,10), we need to identify all vertices (regular points and genus-modifying saddles) from the domain lying on the path from 4 to 10 with function values between 4 and 10 i.e., the nodes {5,6,7.1,8}. A simple method to identify these nodes is a breadth first search traversal starting from 4 in the direction of increasing function value. The traversal terminates when it reaches the vertex 10. The BFS may visit nodes 12 and 14 before reaching the node 10. To avoid including such nodes, we perform a simple comparison test and include a node only if the function value is smaller than 10. The comparison test is suitably modified if the BFS

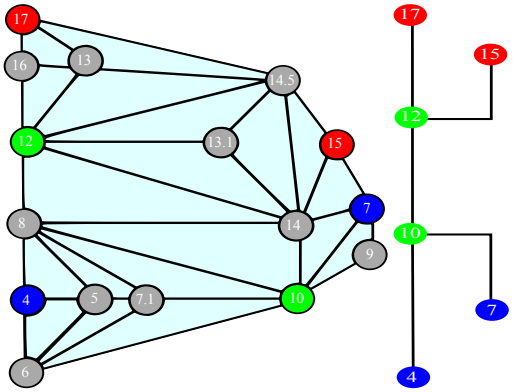


Figure 2: (left) A scalar function sampled at vertices of a triangle mesh and linearly interpolated within the triangle interior. (right) The corresponding contour tree containing 4 leaf nodes, two minima and two maxima, and 2 interior nodes corresponding to saddle points.

originates from a higher vertex and is in the direction of decreasing function value.

The sequence in which the arcs are traversed and the source end point for the BFS traversal need to be carefully chosen to ensure that the procedure remains simple. For example, assume that the arc (10,4) is traversed for augmentation and the BFS originates at the node 10. The traversal cannot be restricted to vertices corresponding to the arc, they may belong to either arc (10,4) or (10,7). So, the traversal always originates at a leaf node.

Let (4,10) be the first arc processed by the algorithm. After augmentation, the arc is marked as processed or removed from the tree. Let (7,10) be the second arc that is augmented. Figure 3 shows the mesh and the contour tree after processing both arcs. All vertices traversed during the two BFS traversals are marked visited. Node 10 is now a leaf node. Next, a BFS originating at node 10 augments the arc (10,12). A BFS traversal that originates from a leaf node necessarily traverses a simply connected region and hence directly identifies all vertices that belong to the incident arc.

3.2 Seed list for traversal

The traversal originating at node 4 terminates when it reaches the node 10. Further, nodes with function value greater than 10 are discarded. So, no additional nodes are included into the arc (4,10). However, the traversal may miss visiting certain nodes. Consider the sub-domain shown in Figure 4, the scalar function defined over it, and the corresponding contour tree.

Assume that the arcs (0,4) and (2,4) are already processed and augmented. Visited vertices are marked. Next, when a BFS traversal begins from vertex 4 to augment the arc (4,10), it terminates immediately because none of the neighbors (vertex 1 and 11) can be visited. Vertex 1 is already visited and vertex 11 is higher than 10. The BFS terminates without visiting vertices 5,6,7,8 that it should have included into the arc. Such a situation may arise at a saddle point, none of whose adjacent vertices may correspond to the arc. Vertex 5 is adjacent to vertices 1 and 11. It is visited during the

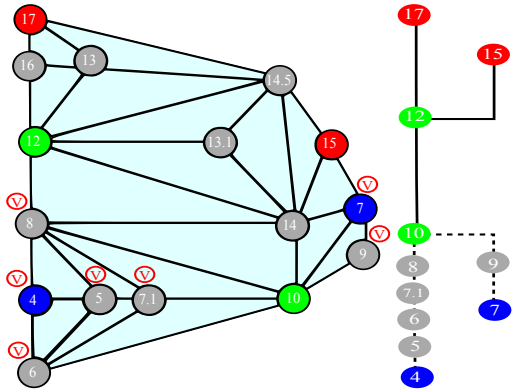


Figure 3: After augmenting arcs (4,10) and (7,10): The triangle mesh and the contour tree are simultaneously traversed in order to identify the degree-2 nodes that belong to each arc of the contour tree. Nodes annotated with an encircled v denote vertices that are already visited. Dashed arcs in the contour tree are already augmented. Traversal beginning from the node 10 will not process these dashed arcs.

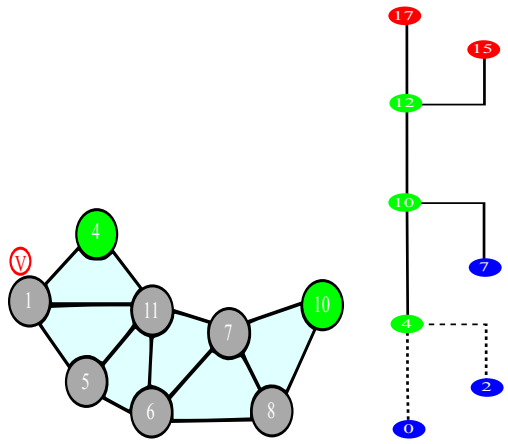


Figure 4: A sub-domain and the relevant arcs from the contour tree. Dashed arcs in the contour tree are already augmented, the degree-2 nodes are not shown here to avoid clutter. Traversal beginning from node 4 along the arc (4,10) will not be able to proceed because its immediate neighbors are either already visited or do not belong to the arc. This situation may arise at saddle points. This necessitates a seed list to be stored at node 4, which can be populated with vertices (5, in this case) while traversing other arcs such as (0,4) that terminate at 4.

traversal from 0 while augmenting the arc (0,4) but discarded because the function value is higher than 4. We propose to maintain a seed list associated with the saddle point 4 that may be used to initialize the BFS queue and begin the traversal. This seed list may be populated with the vertex 5 and other such vertices that are visited while augmenting arcs that end at node 4.

```
Data: AdjD: Adjacency list for domain,
AdjC: adjacency list for contour tree
Result: Out: map with key as arc and value as list of regular
        nodes which belong to that arc
while AdjC is not empty do
    for every node w in AdjC.keys() do
        if w has more than 1 adjacent node in AdjC then
            continue
        end
        start = w
        end = AdjC[start]
        queue = [start] + initialSeedlist of start
        for i in queue do
            visited[i] = true
        end
        while queue is not empty do
            u = queue.pop_front()
            if function value of start < function value of end
              then
                initialSeedlist[end] = initialSeedlist[end] +
                  unvisited neighbors of u in AdjD with higher
                  function value than end
                neighbors = unvisited neighbors of u in AdjD
                  with leser function value than end
            end
            else
                initialSeedlist[end] = initialSeedlist[end] +
                  unvisited neighbors of u in domain with
                  lesser function value than end
                neighbors = unvisited neighbors of u in
                  domain with higher function value than end
            end
            for every n in neighbors do
                queue.push(n)
                visited[n] = true
            end
            if u is not a critical point then
                out[start,end].append(u)
            end
            remove AdjC[start]
            remove start from AdjC[end]
            if AdjC[end] is empty then
                remove AdjC[end]
            end
        end
    end
end
return Out
```

Algorithm 1: Algorithm to augment a contour tree

3.3 Run time Analysis

The algorithm consists of a sequence of modified BFS traversals. Each vertex of the domain is visited once via each incident edge. So, the algorithm runs in O(n) time where n is the number of edges in the input mesh. In practice, the running time is smaller but largely affected by the number of critical points, namely the degree-1 and

degree-3 nodes in the contour tree. A possible reason could be the creation and maintenance of the seed list for the degree-3 nodes.

4 EXPERIMENTAL RESULTS

We report results of computational experiments on 10 different datasets ranging from in size from 3.4K to 1 Million vertices. The contour tree is computed using the open source library ReCon [7]. The augmentation algorithm is implemented in python. The experiments were performed on a workstation with an Intel Xeon Processor E5405 CPU (2.00 GHz base frequency, 12 MB L2 cache) and 8 GB RAM. The running times and memory required to store the augmented contour tree are reported in Table 1.

Datasets testGauss200 and testGauss300 are synthetically generated, by sampling a sum-of-gaussians function on a 2D grid. Other datasets are polygonal models from Visionair's shape repository [19]. For each of the polygonal model, the scalar function value at each vertex is computed as the average distance from 1000 randomly chosen points on the surface. The two columns on the extreme right in Table 1 list the space required to store the contour tree and the augmented contour tree, respectively. The augmented contour tree typically requires at least 10×–20× more space, which can be saved due to the on-demand augmentation. The time required to augment all arcs of the contour tree is reasonable even with a simple python based implementation. It is comparable to run times reported for the sequential version of the algorithm by Gueunet et al. [11]. The running time increases with the number of critical points.

5 APPLICATIONS

We now discuss three different applications that require the augmented contour tree and how the proposed on-demand augmentation will be beneficial.

5.1 Feature-aware selection

The contour tree serves as a useful interface for the user to select features in the data. For example, a node of the contour tree corresponds to a critical point of the scalar function and an arc corresponds to a sub-domain. This sub-domain is a topological feature and is represented by a pair of critical points. During an interactive visual exploration exercise, a user may want to select and highlight a collection of topological features. This is immediately possible if the augmented contour tree is available. More specifically, the specific arcs need to be augmented on-demand.

Figure 5 shows two datasets and the corresponding contour trees. Nodes of the contour tree are ordered along the vertical axis based on their scalar value. One arc is selected (pink) in the contour tree and the corresponding region is highlighted in the domain. In Figure 5(a), the inner contours (white) correspond to the scalar value (4.12) at the maximum contained within the pink region and the outer contour (white) corresponds to the value (3.14) at the other end point saddle of the arc. Note that multiple arcs span the same interval of scalar values as the selected arc. The white contours highlight these "sibling" regions. A long arc is selected for the cat dataset, which corresponds to a region within its tail.

5.2 Isocontour extraction

One of the first steps in the scalar function visualization workflow is isocontour extraction. More specifically, the user may want to extract individual contours. A naive but costly approach is to traverse all cells of the input mesh and identify the cells that contain the required contour. Given the augmented contour tree, we can quickly query the tree to identify the set of arcs of the augmented contour tree that span the input scalar value and trace the contour beginning from the corresponding cells in the domain.

Consider the contour tree in Figure 2. If we want to locate the triangles covered by the level set corresponding to scalar value 8, we first identify the arcs of the contour tree that span the value 8. In this case, we identify two arcs, (4,10) and (7,10). Within each arc, we identify nodes whose scalar value is closest to 8 and hence located a cell (triangle in this case) containing a level set component. Beginning from this cell, we trace the contour using a BFS traversal that explores the neighborhood of the cell to identify the next cell that contains the contour.

Figure 6 shows all contours for a given input scalar value in 3 different datasets. The number of triangles traversed in Figure 6(c) is 1349. This is significantly smaller than the 2.2 Million triangles in the input domain that a naive algorithm will have to traverse in order to extract the isocontour.

5.3 Partial augmentation

The user may often specify a region of interest. In this case, it may be sufficient to augment only certain arcs of the contour tree. The proposed algorithm can be modified to support partial augmentation. In Figure 2, we want to augment the arc (4,10). We could achieve this by invoking a single BFS traversal from the node 4. If we want to augment the arc (10,12), then we need to first traverse arcs (4,10) and (7,10) but not necessarily all arcs in the contour tree.

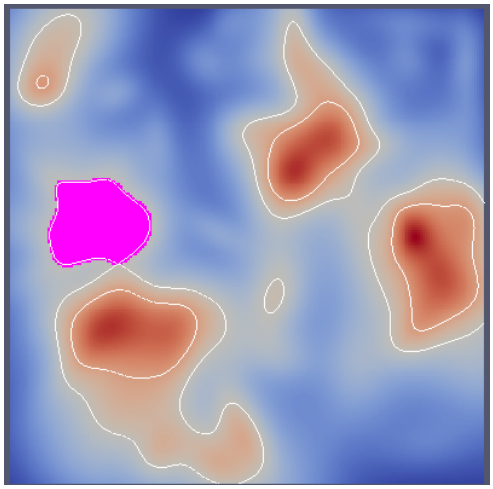
6 CONCLUSIONS

We described an algorithm for computing an augmented version of the contour tree on demand that results in significant savings in terms of storage space. If the number of critical points is small, the augmentation of the entire tree can be computed in a reasonable amount of time using a sequential implementation. The proposed algorithm can be easily modified to augment a select few arcs. In this case, the computation time will be much smaller and the storage space savings will be more significant.

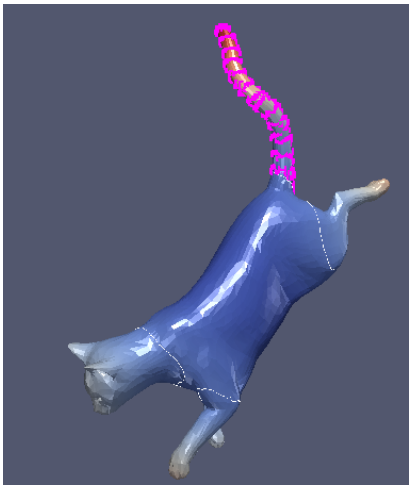
The algorithm may be easily parallelized, arcs incident on maxima and minima may be simultaneously augmented resulting in faster augmentation of the contour tree. The on-demand augmentation of select arcs will enable a software tool for exploration and visualization of a scalar field via interactive selection of arcs in the contour tree.

ACKNOWLEDGMENTS

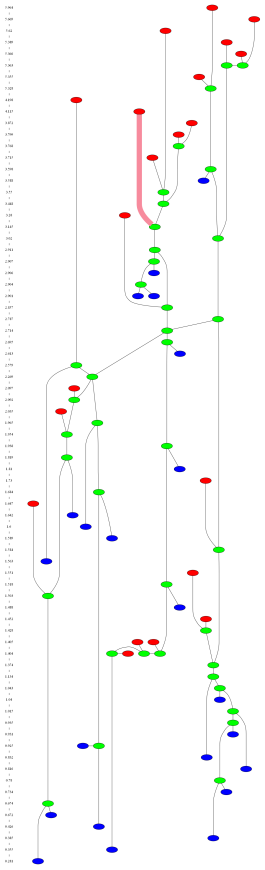
This work is supported by the Department of Science and Technology, India (DST/SJF/ETA-02/2015-16) and the Robert Bosch Centre for Cyber Physical Systems, Indian Institute of Science. We thank Vishnu Nandakumaran for helping generate the contour tree figures.



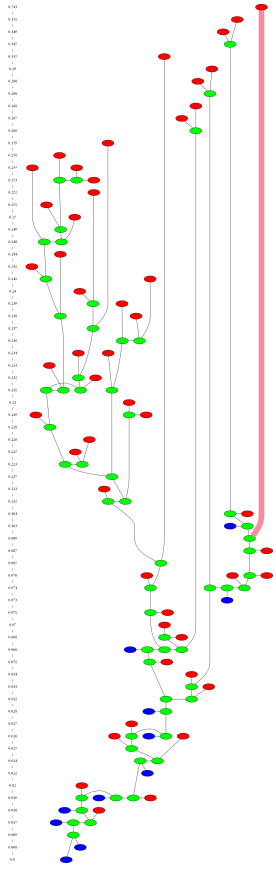
(a) testGauss200



(b) cat



(c) testGauss200 : contour tree

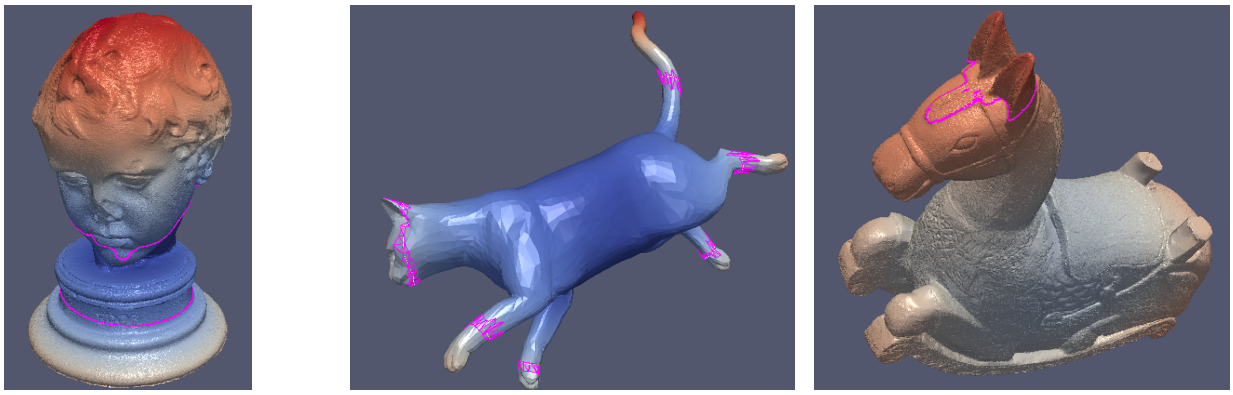


(d) cat : contour tree

Figure 5: Feature-aware selection. Vertices that belong to the sub-domain (a,b) corresponding to a particular arc of the contour tree (c,d) are highlighted. The selected arc and the corresponding region is shown in pink. White contours represent the boundary of regions corresponding to other arcs that span the same scalar values as the selected arc. (c) Arc with end point scalar values {3.14, 4.12} is selected. (b) Arc with end point scalar values {0.09, 0.54} is selected.

Table 1: Time to augment the contour tree for various datasets. The number of vertices and triangles in the domain, number of nodes in the contour tree, time to compute the contour tree, data size, size of the contour tree and the augmented contour tree are reported. We observe that the space required to store the augmented contour tree is much higher than the space required to store the contour tree. [* The time required to compute the contour tree was less than 0.0001ms.]

Dataset	vertices (domain)	triangles (domain)	nodes (contour tree)	contour tree (time(sec))	Aug. tree (time(sec))	Data (size)	contour tree (size)	Aug. tree (size)
cat	3.4K	6.8K	118	0*	0.07	231KB	3.6KB	28.3KB
testGauss200	40K	79K	82	0.26	1.40	4.1MB	2.9KB	351.3KB
testGauss300	90K	179K	88	0.53	4.26	10.2MB	3.2KB	801.4KB
Eros_400K	197K	394K	2223	1.07	8.34	20.1MB	83.6KB	1.9MB
Eros_800K	476K	953K	6013	2.14	29.27	43.9MB	233.4KB	4.8MB
bimba_1Mfaces	502K	1M	928	2.41	32.34	45.5MB	38.6KB	4.9MB
Chinese_dragon	656K	1.3M	16988	2.95	73.57	61.3MB	676.7KB	6.7MB
circular_box	701K	1.4M	95886	4.02	387.00	65.5MB	3.8MB	8.4MB
Ramesses	826K	1.6M	5953	4.29	119.40	77.5MB	235.6KB	8.2MB
Isidore_horse	1.1M	2.2M	16372	5.10	131.82	106.4MB	639.5KB	11.3MB



(a) Eros_800K **(b) cat** **(c) Isidore horse**

Figure 6: Isocontour extraction. An isocontour (pink) intersects a small fraction of the input mesh triangles. The augmented contour tree provides direct access to one triangle intersected by each isocontour component. A simple traversal beginning from this triangle may be used to extract the entire isocontour. (a) Isovalue 0.17, (b) isovalue 0.2, and (c) isovalue 0.41.

REFERENCES

[1] A. Acharya and V. Natarajan. 2015. A parallel and memory efficient algorithm for constructing the contour tree. In *Proc. IEEE Pacific Visualization Symposium*. 271–278.

[2] H. Carr, J. Snoeyink, and U. Axen. 2003. Computing contour trees in all dimensions. *Comput. Geom. Theory Appl.* 24, 2 (2003), 75–94.

[3] Hamish Carr, Jack Snoeyink, and Michiel van de Panne. 2010. Flexible isosurfaces: Simplifying and displaying scalar topology using the contour tree. *Computational Geometry* 43, 1 (2010), 42–58.

[4] Y.-J. Chiang, T. Lenz, X. Lu, and G. Rote. 2005. Simple and optimal output-sensitive construction of contour trees using monotone paths. *Comput. Geom. Theory Appl.* 30, 2 (2005), 165–195.

[5] K. Cole-McLaughlin, H. Edelsbrunner, J. Harer, V. Natarajan, and V. Pascucci. 2004. Loops in Reeb graphs of 2-manifolds. *Disc. Comput. Geom.* 32, 2 (2004), 231–244.

[6] Mark de Berg and Marc J. van Kreveld. 1997. Trekking in the Alps Without Freezing or Getting Tired. *Algorithmica* 18, 3 (1997), 306–323.

[7] Harish Doraiswamy. 2013. ReCon: A fast algorithm to compute Reeb graphs. (2013). http://vgl.csa.iisc.ac.in/software/software.php?pid=003

[8] Harish Doraiswamy and Vijay Natarajan. 2012. Output-Sensitive Construction of Reeb Graphs. *IEEE Trans. Visualization and Computer Graphics* 18 (2012), 146–159. DOI: http://dx.doi.org/10.1109/TVCG.2011.37

[9] Harish Doraiswamy and Vijay Natarajan. 2013. Computing Reeb graphs as a union of contour trees. *IEEE Trans. Visualization and Computer Graphics* 19, 2 (2013), 249–262.

[10] H. Edelsbrunner and J. Harer. 2009. *Computational Topology: An Introduction*. Amer. Math. Soc., Providence, Rhode Island.

[11] Charles Gueunet, Pierre Fortin, Julien Jomier, and Julien Tierny. 2017. Task-based augmented merge trees with Fibonacci heaps. In *Large Data Analysis and Visualization (LDAV), 2017 IEEE 7th Symposium on*. IEEE, 6–15.

[12] Aaditya G Landge, Valerio Pascucci, Attila Gyulassy, Janine C Bennett, Hemanth Kolla, Jacqueline Chen, and Peer-Timo Bremer. 2014. In-situ feature extraction of large scale combustion simulations using segmented merge trees. In *Proc. ACM/IEEE Conf. on Supercomputing (SC14)*, Vol. 14.

[13] Senthilnathan Maadasamy, Harish Doraiswamy, and Vijay Natarajan. 2012. A hybrid parallel algorithm for computing and tracking level set topology. In *High Performance Computing (HiPC), 2012 19th International Conference on*. IEEE, 1–10.

[14] Dmitriy Morozov and Gunther Weber. 2013. Distributed merge trees. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*. ACM, 93–102.

[15] Dmitriy Morozov and Gunther H Weber. 2014. Distributed Contour Trees. In *Topological Methods in Data Analysis and Visualization III*. Springer, 89–102.

[16] Valerio Pascucci and Kree Cole-McLaughlin. 2003. Parallel Computation of the Topology of Level Sets. *Algorithmica* 38, 1 (2003), 249–268.

[17] Valerio Pascucci, Giorgio Scorzelli, Peer-Timo Bremer, and Ajith Mascarenhas. 2007. Robust on-line computation of Reeb graphs: simplicity and speed. *ACM Trans. Graph.* 26, 3 (2007), 58.

[18] G. Reeb. 1946. Sur les points singuliers d'une forme de Pfaff complètement intégrable ou d'une fonction numérique. *Comptes Rendus de L'Académie ses Séances, Paris* 222 (1946), 847–849.
[19] Aim@Shape: Digital shape workbench. 2011. The Shape Repository. (2011). http://visionair.ge.imati.cnr.it/ontologies/shapes/
[20] Sergey P. Tarasov and Michael N. Vyalyi. 1998. Construction of contour trees in 3D in O(n log n) steps. In *Proceedings of the fourteenth annual symposium on Computational geometry (SCG '98)*. ACM, New York, NY, USA, 68–75.
[21] Dilip Mathew Thomas and Vijay Natarajan. 2011. Symmetry in scalar field topology. *IEEE Trans. Visualization and Computer Graphics* 17, 12 (2011), 2035–2044.
[22] Tony Tung and Francis Schmitt. 2004. Augmented Reeb Graphs for Content-Based Retrieval of 3D Mesh Models. In *SMI '04: Proc Shape Modeling Intl.* 157–166.
[23] M. van Kreveld, R. van Oostrum, C. Bajaj, V. Pascucci, and D. R. Schikore. 1997. Contour trees and small seed sets for isosurface traversal. In *Proc. Symp. Comput. Geom.* 212–220.
[24] M. van Kreveld, R. van Oostrum, C. Bajaj, V. Pascucci, and D. R. Schikore. 1998. *Contour trees and small seed sets for isosurface traversal.* Technical Report UU-CS-1998-25. Department of Computer Science, Utrecht University.
[25] G. H. Weber, S. E. Dillard, H. Carr, V. Pascucci, and B. Hamann. 2007. Topology-controlled volume rendering. *IEEE Trans. Visualization and Computer Graphics* 13, 2 (2007), 330–341.
[26] Jianlong Zhou and Masahiro Takatsuka. 2009. Automatic Transfer Function Generation Using Contour Tree Controlled Residue Flow Model and Color Harmonics. *IEEE Trans. Visualization and Computer Graphics* 15, 6 (2009), 1481–1488.