W-Structures in Contour Trees

Petar Hristov and Hamish Carr

Abstract The contour tree is one of the principal tools in scientific visualisation. It captures the connectivity of level sets in scalar fields. In order to apply the contour tree to exascale data we need efficient shared memory and distributed algorithms. Recent work has revealed a parallel performance bottleneck caused by substructures of contour trees called W-structures. We report two novel algorithms that detect and extract the W-structures. We also use the W-structures to show that extended persistence is not equivalent to branch decomposition and leaf-pruning.

1 Introduction

Topology is the basis for persistent homology [17], a framework for extracting structural information from data, and Computational Topology [35], which studies how to compute and scale topological structures efficiently. Topological algorithms and data structures have been applied to various problems in structural biology [14, 34], computer vision [3, 23] medical imaging [32] and visualisation [5, 6, 10].

The Contour Tree (CT) is a data structure that captures the topological connectivity of a scalar field. In scientific visualisation it is used to identify features of more than local importance in large scale scientific and engineering simulations [22, 30]. As the size of data sets grows to exascale there is an increasing demand to develop scalable massively multicore and distributed algorithms and systems.

Electronic supplementary material The online version of this chapter (https://doi.org/10.1007/978-3-030-83500-2_1) contains supplementary material, which is available to authorized users.

P. Hristov (✉) · H. Carr
University of Leeds, Leeds LS2 9JT, UK
e-mail: mm16pgh@leeds.ac.uk

H. Carr
e-mail: h.carr@leeds.ac.uk

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2021

I. Hotz et al. (eds.), *Topological Methods in Data Analysis and Visualization VI*, Mathematics and Visualization, https://doi.org/10.1007/978-3-030-83500-2_1

The contributions of this paper are to extend the understanding of a pathological case, called the *W-structure* [12], that emerges in one of the state of the art parallel contour tree algorithm (Subsect. 2.2); to demonstrate that W-structures have implications for parallel algorithmic efficiency (Subsect. 2.2) and that they can be detected and characterized (Subsect. 3.2); and finally that they can be used to show that contour tree simplification via persistent homology and branch decomposition [29] are equivalent (Sect. 5).

2 Background

The goals of this work are study how W-structures affect parallel contour tree computation and how branch decomposition relates to persistent homology. The relevant background is split between Sect. 2 and Sect. 5 because these two tasks have different prerequisites. In this section, we introduce methods to compute (Subsect. 2.2) and simplify (Subsect. 2.3) the Contour Tree (Subsect. 2.1) and leave the background relating to persistent homology to Subsect. 5.1.

2.1 Contour Trees

Given a scalar function $f : \mathbb{R}^n \to \mathbb{R}$, a *level set* is the set of all points with a given *isovalue* h: $f^{-1}(\{h\}) = \{x \in \mathbb{R}^n \mid f(x) = h\}$. We refer to individual connected components of a level sets as *contours*. As h varies, contours may appear, disappear, connect or disconnect at *critical points* where the gradient vector is zero.

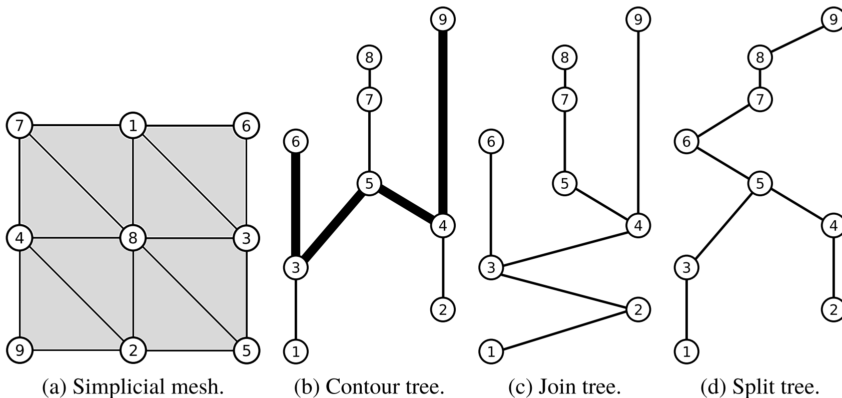


Fig. 1 A simplicial mesh **a** that generates a W-structure, the corresponding contour tree **b** with the W-structure shown as thicker edges and the corresponding join and split trees **c** and **d**. The vertices are labeled with their height value

The Reeb Graph of f is constructed by contracting the contours at every isovalue to a point. The resulting structure is a graph whose vertices are critical points and whose edges are families of contours with identical connectivity. Since $\mathbb{R}^n$ is simply connected, the Reeb Graph is connected and acyclic, also called the Contour Tree [4].

We extend the definition of a level set to a super-level set: the points with higher values than h, i.e. $\{x \in \mathbb{R}^3 \mid f(x) \geq h\}$, or a sub-level set with lower values, $\{x \in \mathbb{R}^3 \mid f(x) \leq h\}$. Contracting the connected components of the super-level and sub-level set gives two *merge trees*, also referred to as the join & split trees.

2.2 Contour Tree Algorithms

In practice, we assume that the domain is approximated by a simplicial mesh with a linear interpolant. Under these constraints, critical points occur at vertices, and we only need to process the graph composed of the vertices and edges of the mesh. Although these assumptions can be relaxed [7], they are the most common in practical data analysis, and they simplify our analysis without loss of generality.

The standard contour tree algorithm [8] is based on the idea of an isovalued sweep - i.e. processing the vertices of the mesh in sorted order from high to low. As each vertex u is processed, any edge (u, v) to a higher-valued vertex v is also processed. At each step there is a subgraph representing the super-level set, whose connectivity can be tracked with an incremental version of the union-find data structure [33]. In the first stage of the algorithm we construct the join tree, then repeat with a low-to-high sweep to compute the split tree. In the second stage, we construct the contour tree iteratively by transferring leaves and their adjacent edge from the merge trees, using induction on a simple invariant to guarantee correctness. As a result, this algorithm is sometimes referred to as the *sweep and merge* algorithm.

There are several approaches to scaling sweep and merge. Distributed algorithms [22, 26–28] adopt a divide and conquer approach where each node computes the contour/merge tree on parts of the data. The scalability of distributed methods relies not only on minimising node communication but also efficient utilisation of individual nodes. Efficient utilisation of nodes relies on parallel algorithms using vector [12], thread [19, 20] or hybrid [2, 24, 31] shared memory parallelism.

The parallel peak pruning algorithm [12] is the only shared memory algorithm which parallelises the merge phase. Other algorithms introduce a novel way of computing the join and split tree, but combine them in serial. Note that the merge phase has linear complexity and it is significantly faster to compute than the join and split trees. Nonetheless, parallelising the merge phase is important for resource utilisation and parallel speed up according to Amdahl's law [20].

In the merge phase of the serial contour tree algorithm [8] we transfer the leaves and their adjacent edge from the merge trees to the contour tree. Since this is a local operation all leaves can be batched and transferred in a single parallel step. The algorithm alternates between transferring leaves from the join and split tree until the contour tree is fully constructed. In the ideal case, each batch transfers at least half

of the vertices, guaranteeing logarithmic performance. In a tree with no vertices with degree two, half of the vertices are leaves. Thus we can achieve logarithmic collapse if we remove degree two vertices in a post process for each batch.

Removing a degree two vertex is straightforward when its neighbouring vertices have values spanning the value of the vertex. This is the case when the vertex is connected by a chain of such vertices to a leaf. In effect, these vertices are regular at this stage (although they may not have been in earlier stages), and can be removed. For vertices of degree two whose value is smaller or bigger than the value of both neighbours, this is not so easy to perform. We call these vertices forks.

A *W-structure* consists of repeated forks zigzagging between upwards and downwards, as illustrated in Fig. 1. In order to collapse the W-structure completely we can only prune from an endpoint of the W-structure to a fork. The internal vertices cannot be process until we have pruned all forks. Therefore computation is effectively serialized along the largest W-structure in the contour tree. This prevents logarithmic collapse and complicates the parallel complexity analysis of the algorithm.

W-structures were first visible in Carr et al. [9], they have previously complicated proofs [11] and have caused issues with contour tree parallel algorithm design and analysis [12]. The contribution of this paper is to initiate a systematic study of these W-structures. A natural starting point is to describe them mathematically and develop algorithms to detect and extract them from contour trees. These algorithms will allow us to quantify the impact they have on computation and determine whether they are an issue in real life data sets. Finally we will show that W-structures lead to theoretical complications as well, by demonstrating that contour tree simplification via branch decomposition is not equivalent to persistent homology.

2.3 Contour Tree Simplification

The principal technique for contour tree simplification is branch decomposition [29]. The contour tree is partitioned into a set of disjoint monotone paths (branches). A branch decomposition is hierarchical when there is exactly one branch that connects two leaves called the master branch and every other branch connects a leaf to a saddle. Branches represent pairs of critical points that can be cancelled [25].

In order to decide the order of cancellation we use the persistence of the branches [29]. The persistence of a branch is the greater of the difference between the height value at its endpoints and the persistence of its children. Simplification consists of removing branches that do not disconnect the tree in order of their persistence. This produces a hierarchy of cancellations as shown in Fig. 2. In branch decomposition we repeatedly pair upper leafs to join saddles and lower leaves to split saddles in order of persistence until all vertices are paired.

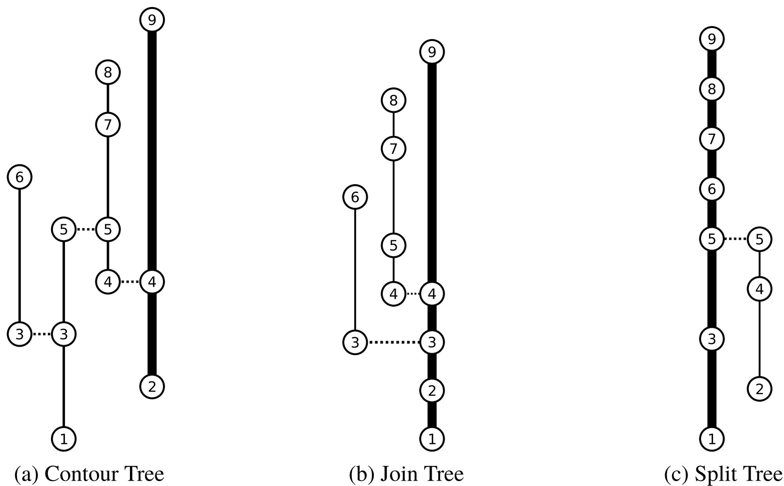
As an example we consider branch decomposition of the contour tree from Fig. 1 (b). The first two candidate branches are $(6, 3)$ with persistence 3 and $(4, 9)$ with persistence 5. We take the branch with lower persistence $(6, 3)$. In the next step the candidates are $(1, 5)$ with persistence 4 and $(4, 9)$ with persistence 5. We take

(1, 5). The remaining candidate branches are (4, 8) with persistence 4 and (4, 9) with persistence 5. After removing (4, 8) the only remaining branch is the master branch - (2, 9). To conclude, the pairs of critical points produced by the branch decomposition of the contour tree are (3, 6), (1, 5), (4, 8) and (2, 9).

We can also compute the branch decomposition of the join and split trees. The candidate branches for the join tree are (3, 6), (4, 8) and (4, 9). We remove them in order of persistence. First (3, 6), then (4, 8) and finally the master branch (1, 9). In the split tree the two candidate branches are (1, 5) and (2, 5). We remove (2, 5) first because it has lower persistence and then the master branch (1, 9).

3 W-Structures in Contour Trees

In this section we will develop the existing understanding about W-structures and introduce three algorithms to compute the largest W-structure in a contour tree. The value of these algorithms will be in that they will allow us to build our understanding of W-structures and allow us to detect W-structures in real life data. We adopt the following notation: in a contour tree T the set of vertices is V and the set of edges is E. We refer to paths in the contour tree by their first and last vertex because there is a unique path between any two vertices.



(a) Contour Tree (b) Join Tree (c) Split Tree

Fig. 2 Branch decomposition of the contour tree and the two merge trees from Fig. 1 with the edges of the master branches in thicker lines. Vertices are labeled with their height. In both merge trees the master branch is the monotone path from the global minimum 1 to the global maximum 9. Note that in the absence of a monotone path between 1 and 9 in the mesh and its contour tree they cannot be paired

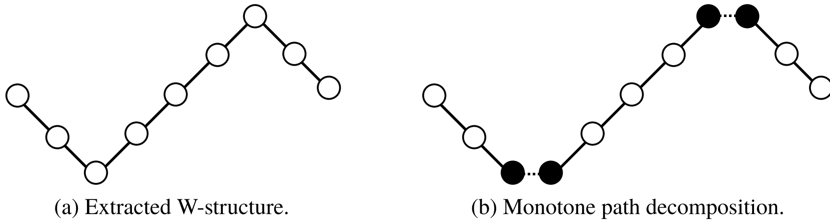


Fig. 3 A W-structure and its monotone path decomposition (forks in solid black)

3.1 Spatial Characterization

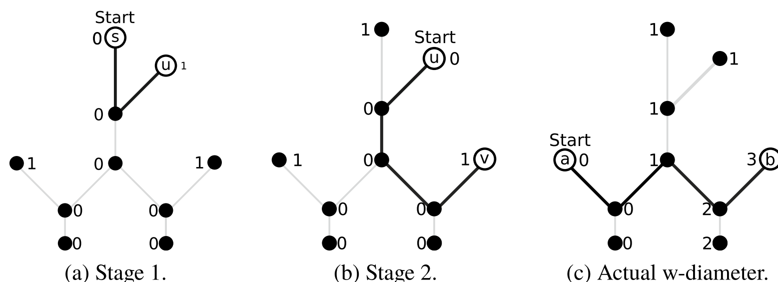
An important property of paths in contour trees is their monotone path decomposition. This is a sequence of monotone subpaths that share exactly one vertex and have alternating direction (Fig. 3). We can use the number of subpaths in the monotone path decomposition to characterize them. To simplify this characterization we note that the number of subpaths in the monotone path decomposition is one more than the number of vertices where an ascending subpath ends and a descending subpath begins (or vice versa). We will call these vertices forks.

We define the w-length of a path as the number of forks in that path and the length of a path as the number of edges. To avoid ambiguity between w-length and length we will use the term w-path to emphasize that we are referring to a path's w-length. Note that if two paths share a vertex it may be a fork in one of them, but not the other. For example vertex 5 from the contour tree in Fig. 1 is a fork in the path from 6 to 9, but not in the path from 1 to 8. This property is crucial in understanding how we develop algorithms for detecting W-structures.

In this terminology the largest W-structure in a contour tree is a path between two leaves with maximum w-length (or longest w-path). We will call this the *w-diameter* of the contour tree. This again is analogous to how the longest path in a tree is called the diameter of the tree. As there are efficient algorithms for computing the diameter of a tree a natural question to ask is whether we can adapt these algorithms to compute the w-diameter of a contour tree.

3.2 W-Diameter Algorithms

We will begin the development of w-diameter algorithms by first describing three tree diameter algorithms—brute force, endpoint search and root based search. The brute force algorithm computes the lengths of all paths in a tree and outputs the maximum one. The second algorithm relies on the fact that the most distant leaf from any vertex is the endpoint of the diameter [15]. It requires two breadth first searches, so it has linear running time. The third algorithm is defined for rooted trees where the longest path may or may not pass through the root. If it passes through the root then it must



(a) Stage 1. (b) Stage 2. (c) Actual w-diameter.

Fig. 4 Execution of the Double BFS algorithm **a** and **b** on an example contour tree and the actual w-diameter **c**. Black edges indicate a path of maximum w-length. Numerical labels next to vertices indicate their w-distance from the start vertex. In stage 1 we start from an arbitrary vertex s and find the most w-distant vertex from it u. In stage 2 we find the most w-distant vertex from u and call it v. The w-length of the path from u to v is however suboptimal as demonstrated in the last figure. If our initial root was a then the algorithm would have obtained the w-diameter

start in a leaf in one of the subtrees of a child of the root, pass through the root and end in a leaf in another subtree. If it does not then it must be entirely contained in a subtree of a child of the root. With the use of dynamic programming this algorithm has linear complexity as well. In the next three subsections we will adapt each one of these algorithms to compute the w-diameter of a contour tree.

3.3 Algorithm 1—Multi BFS

The brute force approach to finding the w-diameter of a contour tree compares the w-lengths of all paths in the contour tree. To implement it we modify Breadth First Search (BFS) to traverse the tree and compute w-length (number of forks) instead of length (number of edges). We then run this modified BFS from every vertex in the tree and output the maximum value found. It has quadratic running time and we will refer to it as Naive BFS.

3.4 Algorithm 2—Double BFS

The algorithm works the same as the second tree diameter algorithm we described in Subsect. 3.2 except we measure w-distance instead of distance. Consider a contour tree T and an arbitrary start vertex s. First we find the most w-distant vertex from s and call it u. Then we find the most w-distant vertex from u and call it v. After the first search from s we are not guaranteed that v is an endpoint of a w-diameter. We are however guaranteed that v is an endpoint of a path whose w-length is at least that of the w-diameter minus two. We will demonstrate why that is true in the following two paragraphs.

We illustrate this with an example in Fig. 4. In this example the algorithm does not produce the w-diameter of the contour tree which is the path from a to b. The mismatch between the output of the algorithm and the actual w-diameter is due to vertices which are forks in one path, but not in others. First consider the black vertex on the path from s to u in Fig. 4 a); it is a fork on the path from s to u, but not on the path from u to $v = b$ (Fig. 4 b). Secondly, consider the midpoint of the path from a to b; it is a fork on the path from a to b, but not on the path from u to b or a.

Our argument for the general case is based on this example. In a contour tree T let s be the start vertex and v the most w-distant vertex from s. During the course of the algorithm there are two graph searches, one from s to identify v and one from v. There are at most two turning points which may or may not be forks during the two searches. One of the turning points is where the path from s to a (or symmetrically to b) diverges from the path from s to v. The other one is where the path from v to a (or symmetrically to b) diverges from the path from a to b (or symmetrically from b to a). This causes the w-length of the path we find to vary by at most two from the w-diameter of the contour tree.

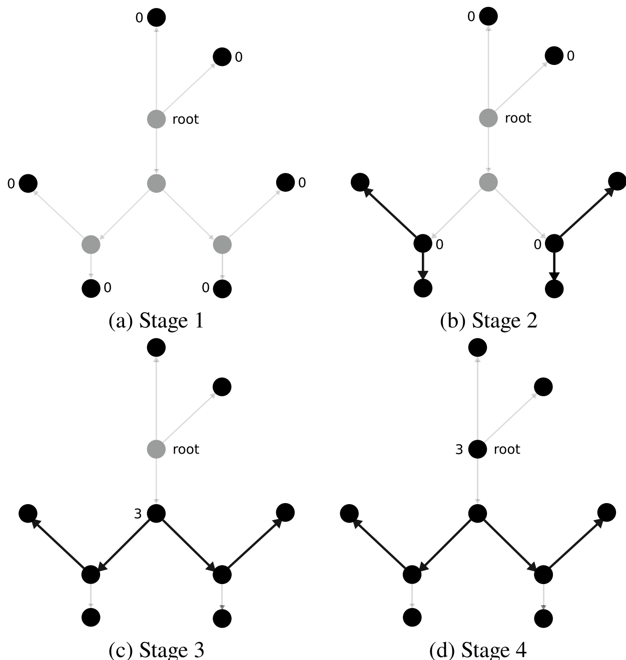
To implement Double BFS we pick a starting vertex and then run the modified Breadth First Search twice. The first BFS from the root to find the farthest leaf from it and then a second BFS from that leaf. The algorithm consists of two consecutive Breadth First searches and therefore its running time is $O(|V|)$.

3.5 Algorithm 3—Dynamic

The third algorithm works by progressively combining paths from subtrees of the contour tree to obtain the longest w-path. For a contour tree T we pick an arbitrary start vertex s to be the root of the rooted tree T_s. Observe that the w-diameter of T either passes through s or it does not. If it does pass through s then it must also pass through two children of s and be contained in their subtrees. If it does not pass through s then it must be entirely contained in the subtree of one of the children of s. We can then extend this reasoning recursively to all the subtrees of T_s.

For every vertex u in T_s we define $T_{s,u}$ as the subtree of T_s with root u. We find the w-diameters of all subtrees of $T_{s,u}$ and use them to compute the w-diameter of $T_{s,u}$. We now demonstrate how to compute the w-diameter of $T_{s,u}$ assuming that the optimal solutions for all subtrees of $T_{s,u}$ have been found recursively. Note that the base case is at the leaves of the tree Fig. 5 a).

Case 1—the w-diameter of $T_{s,u}$ goes through u. To handle this case we must find two maximum paths contained in two subtrees whose roots are children of u, say a and b. As we have recursively found all such maximum paths we only have to determine how to combine them. When combining them *three* vertices can become forks. The first one is u and the other two are a and b which were previously endpoints of the maximum paths in their subtrees. To account for u we simply have to compare



(a) Stage 1 (b) Stage 2

(c) Stage 3 (d) Stage 4

Fig. 5 Execution of the Dynamic algorithm on an sample contour tree. Black vertices have been processed, gray ones have not. The black edges are the w-diameters for all subtrees. The numerical label next to a vertex indicates that it is being processed in the current stage and the number is the w-diameter in its subtree. Each stage combines w-diameters of smaller subtrees to obtain the diameter of the whole tree

its height to a and b. To account for a and b we must compare their height with u and the previous vertex in the maximum w-path they are the endpoint of.

An example of where Case 1 holds is in Fig. 5 c). Note that this requires us to not only look at all children of u, but also to all children of children of u. In addition to this it may be the case that u is a leaf and has only one child say a. In this case u must be the endpoint of the w-diameter. In this case we find all maximum paths that end at a and account for whether a becomes a fork in them or not.

Case 2—the w-diameter of $T_{s,u}$ does not pass through u. In this case the w-diameter has to be entirely contained in one of the subtrees whose root is a child of u so we pick the maximum one. An example of where Case 1 holds is in Fig. 5 d).

Let us derive the time complexity of the algorithm. Firstly it takes $O(|V|)$ time to traverse the tree and root it via either BFS or DFS. Secondly, we iterate over the children of all children of all vertices. Since the algorithm operates on trees, every vertex has a unique grandparent. Therefore every vertex will be visited exactly once and contribute $O(|V|)$. Finally we pick all pairs of children of a vertex to find the maximum w-path that goes through the vertex. Computing this for all vertices in the graph yields $O(\sum_{u \in V} d(u)^2)$ where $d(u)$ is the degree of a vertex. To see how we can

evaluate this consider that in a tree $d(u) + d(v) \leq |V|$ for any two vertices connected by an edge (otherwise we would have a cycle). If we sum over all edges we obtain that $\sum_{uv \in E(T)} d(u) + d(v) \leq |V|^2$. In the summation on the left hand side every term $d(u)$ is present $d(u)$ times and therefore $\sum_{uv \in E(T)} d(u) + d(v) = \sum_{u \in V(T)} d(u)^2 \leq |V|^2$. Therefore the overall complexity of the algorithm is $O(|V|^2)$. Note that this is formally more complex than the dynamic programming tree diameter algorithm in Sect. 3.2. The difference between the two is that the base algorithm does not need to look at all pairs of children, it can simply pick the child with the largest height.

We have shown that the algorithmic complexity of the Dynamic algorithm is no better than that of the Naive BFS algorithm. However, the running time of the Dynamic algorithm depends on the average degree of the vertices of the contour tree. If we assume the function is generic and PL Morse [16], then the degree of every vertex is bounded and the running time of the algorithm is linear.

4 Empirical Study

In this section we supplement the theoretical investigation of the W-structures with an empirical study. We verify the correctness and running time of the w-diameter algorithms and study the W-structures in contour trees of real life data sets.

We implemented all three w-diameter algorithms in C++. Their source code is in the supplementary materials. We avoided a recursive implementation of the Dynamic algorithm due to excessive overhead caused by recursive calls. Instead we traversed the tree once with a standard Breadth First Search to identify the children and parents of all vertices. We then put them in a list and starting from the leaves we processed all vertices in the tree making sure their children are processed beforehand.

The data sets we have used are taken from the Open SciVis Dataset [1]. The contour tree was computed using the open source VTK-m implementation. All tests were run on a Lenovo E550 Laptop with Intel(R) Core(TM) i5-5200U CPU at 2.20 GHz and 8 GB DDR3 RAM at 1600 MHz. The running time of the w-diameter programs was obtained from an average of five runs on each data set.

4.1 Results

The results from the empirical tests are shown in Table 1. The first column shows the name of the data set, the second column the number of vertices and the third the diameter of the contour tree (not w-diameter). The fourth column shows the number of iterations in the merge phase of the PPP algorithm. The following columns show the running times and output of the Double BFS, Dynamic and Naive BFS algorithms.

From previous work [12] we know that in a contour tree without W-structures the parallel step complexity of the merge phase is logarithmic. When W-structures are

Table 1 Analysis of the W-Diameter of real life data sets

Dataset	Vertices	Tree	Merge	Double BFS		Dynamic		Naive BFS	
		Diam.	Iter.	Time(s)	W-Diam	Time(s)	W-Diam	Time(s)	W-Diam
fuel	236	43	6	0.0001	2	0.0003	4	0.0133	4
marschner lobb	1604	371	9	0.0008	3	0.0022	3	0.3881	3
hydrogen atom	13038	3153	6	0.0039	2	0.0139	4	25.382	4
aneurism	65625	23701	10	0.0251	4	0.0671	4	723.39	4
engine	518780	92559	12	0.2183	6	0.5504	6	N/A	N/A
foot	870371	133655	14	0.4922	7	1.0342	7	N/A	N/A
skull	2199876	340611	14	1.0839	7	2.5048	7	N/A	N/A
backpack	7441922	1365783	18	4.2384	7	8.4635	7	N/A	N/A
bunny	13078906	1450364	18	8.1435	5	15.903	6	N/A	N/A
present	17006950	2349226	16	11.339	5	21.075	6	N/A	N/A
christmas tree	24643034	4866458	17	13.399	5	29.015	5	N/A	N/A
magnetic rec.	40321359	6401594	18	34.480	6	57.287	6	N/A	N/A

taken into account the best formal guarantee that could be given is the tree diameter. However it was noted that the merge phase usually takes less than a logarithmic number of iterations. This is reflected in our results as well - the number of iterations is always less than $\log_2(|V|)$ and substantially less than the diameter.

In this paper we investigate the case where W-structures are present in the contour tree. The key issue in doing so is to detect when they are present and to quantify their size. Based on the results of this empirical study we can confirm that W-structures do appear in real world data. Fortunately, the size of the W-structures is relatively small compared to the size of the data. Furthermore larger data sets do not seem to have a proportionally larger w-diameter.

A w-diameter of more than $2\log_2(|V|)$ can formally prevent logarithmic collapse in the merge phase. The maximum w-diameter in our tests was 7 which is less than $2\log_2(|V|)$ in all cases. This highlights the importance of parallelising the merge phase and explains why it performs well in practice.

Finally the running time and correctness of the w-diameter algorithms is consistent with our theory. The worst case running time for Dynamic is quadratic, but as predicted it is not exhibited and in practice it is only around twice as slow as the Double BFS algorithm. The Dynamic and Naive BFS algorithms produce the same results, while the Double BFS algorithm produces a suboptimal w-path in the bunny, present, fuel and hydrogen atom data sets.

5 W-Structure Simplification

The topological complexity of a scalar field is largely governed by the number of critical points. Some methods for simplification remove critical points in pairs using an auxiliary topological data structure such as the Morse-Smale Complex, persistent homology or the Contour Tree. Once selected, pairs are ranked by persistence, or more advanced metric such as volume or surface area [9], and cancelled in that order.

It is well know that the sequence of simplifications do not always agree with persistence. For example in Morse-Smale complexes there are blocking structures known as strangulations [21] and in ϵ-simplification [18] compound function value changes can occur when following the persistence order. W-structures are a similar kind of blocking structures in the case of contour tree simplification.

What is less well known is the relationship between branch decomposition and persistent homology (first observed [29]). In the merge phase, the contour tree is build up of branches taken from the join and split tree. The critical pairs defined by those branches in turn correspond to the 0th persistence pairs of f and $-f$. In this chapter we will consider whether the branch decomposition of the contour tree also corresponds to those persistence pairs.

5.1 *Persistent Homology Overview*

The building blocks of persistent homology [17] are sequences of nested simplicial complexes called filtrations. In a filtration we start from the empty set and iteratively add simplices to obtain the full complex. Throughout this section will consider the ascending and descending filtrations on Fig. 6 of the simplicial mesh from Fig. 1. The ascending filtration of M is made up of the sub-level sets M_i which contain all vertices whose value is lower than or equal to i and all other simplices between them. The descending filtration of M is made up of the super-level sets M^i which contain all vertices whose value is bigger than or equal to i and all other simplices between them. In short, this is exactly the same as the join and split tree computation in the sweep and merge algorithm we discussed in Subsect. 2.2.

Persistent homology describes how the n-dimensional connectivity of the simplicial complexes in a filtration changes. The n-dimensional connectivity is described by an algebraic structure called the nth homology group. Since contour and merge trees only capture connected components and not higher dimensional connectivity such as holes and voids we will only need to consider the 0th homology group. When a connected component appears in the progression of the filtration we say that a 0th homology class is *born*. When two connected components merge together the 0th homology class that corresponds to the younger component *dies* and the 0th homology class that corresponds to the older one *persists*.

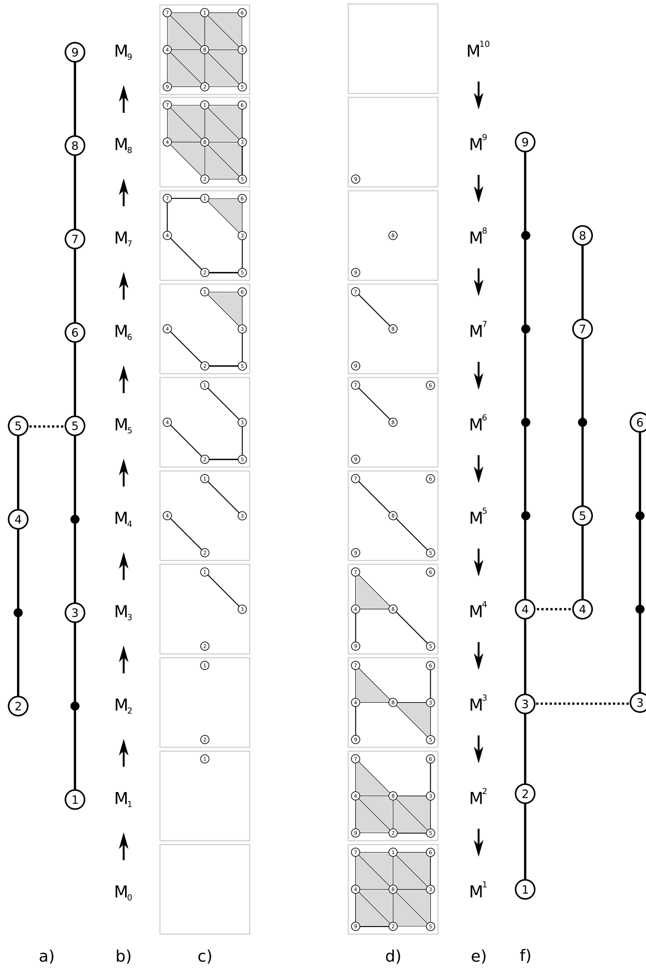


Fig. 6 Ascending **c** and Descending **d** filtration of Fig. 1 **a** Direction of travel of the two filtrations **b**and **d**. Branch decomposition of the split **a** and the join tree **f** with additional vertices corresponding to connected components

The output of persistent homology is the so called persistence pairs. A persistence pair is a record of the birth and death of a 0th homology class. The persistence pairs of a filtration give a basis for topological simplification.

5.2 Comparison of Critical Point Pairs

The proposition we aim to resolve is whether persistence pairs are equivalent to branch decomposition cancellation pairs. Since monotone paths in the contour tree correspond to monotone paths in the simplicial mesh [8] then branches obtained via branch decomposition correspond to valid topological cancellations in the simplicial mesh [29]. Therefore they are comparable to cancellations given by persistence pairs.

We start by computing the persistence pairs of the ascending filtration in Fig. 6 c). One connected component appears in the complex M_1, another one in the complex M_2 and they merge in M_5. When the two merge the older one born in M_1 persists and the younger one born in M_2 dies. We record this with the persistence pair $(2, 5)$. When the filtration is done we are left with a 0th homology class corresponding to the single connected component of the mesh. This 0th homology class is called essential and we give it an infinite persistence pair $(1, \infty)$.

In the descending filtration in Fig. 6 d) we can see that three 0th homology classes are born in the complexes M^9, M^8 and M^6. The one born in M^8 dies in M^4 when it merges with the one born in M^9 (because M^9 is older). The one born in M^6 dies in M^3 when it merges with the one born in M^9. The one born in M^9 does not die in the descending filtration because it represents the connected component of M itself. Therefore the persistence pairs are $(6, 3)$, $(8, 4)$ and $(9, \infty)$.

The two infinite persistence pairs can be resolved with extended persistence [13]. The idea behind extended persistence is that we take the essential classes from the ascending and descending pass and pair those which correspond to the same connected component. More generally we know that in a simple domain extended persistence always pairs the global minimum with the global maximum. In our example the extended persistence pair for the ascending filtration is $(1, 9)$ and the extended persistence pair for the descending filtration is $(9, 1)$.

Finally consider the branch decomposition of the contour tree in Fig. 2. While the first produced branch $(6, 3)$ is the same in the contour tree and in the join tree branch decomposition, the third branch of the join tree branch decomposition $(1, 9)$ does not occur as a pair in the contour tree. The same holds for the branch decomposition of the split tree and the persistent homology pairs of the ascending and descending filtration - they all pair the global minimum 1 with the global maximum 9. This cannot occur in the branch decomposition of the contour tree because branches represent monotone paths. There is no monotone path between the 1 and 9 in the mesh and therefore no monotone path in the contour tree. The result then follows.

6 Conclusion

In this paper we introduced the theory of a pathological case in contour tree parallel computation called a *W-structure*. We developed three algorithms to detect and extract W-structures and showed that they appear in real life data. We also showed

that W-structures cause fundamental theoretical issues. They lead to an example that contour tree simplification is not equivalent to persistent homology.

Future work in this direction will focus on whether there is a form of persistent homology which matches the branch decomposition form of simplification, on algorithmic improvements for which W-structures do not pose a parallel bottleneck, and if need be, on further empirical studies to inform algorithmic development.

Acknowledgements This work was supported by the University of Leeds School of Computing, including a scholarship supporting the first author. We would also like to thank Ingrid Hotz, Talha Bin Masood, Filip Sadlo and Julien Tierny for organising the TopoInVis 2019 workshop.

References

1. Open SciVis Datasets. https://klacansky.com/open-scivis-datasets. Accessed 30 Jan 2020
2. Acharya, A., Natarajan, V.: A parallel and memory efficient algorithm for constructing the contour tree. In: 2015 IEEE Pacific Visualization Symposium (PacificVis), pp. 271–278 (2015). https://doi.org/10.1109/PACIFICVIS.2015.7156387
3. Biasotti, S., Giorgi, D., Spagnuolo, M., Falcidieno, B.: Reeb graphs for shape analysis and applications. Theor. Comput. Sci. **392**(1–3), 5–22 (2008)
4. Boyell, R.L., Ruston, H.: Hybrid techniques for real-time radar simulation. In: Proceedings of the 1963 Fall Joint Computer Conference, pp. 445–458. IEEE (1963)
5. Bremer, P.T., Hamann, B., Edelsbrunner, H., Pascucci, V.: A topological hierarchy for functions on triangulated surfaces. IEEE Trans. Visual. Comput. Graph. **10**(4), 385–396 (2004)
6. Carr, H., Geng, Z., Tierny, J., Chattopadhyay, A., Knoll, A.: Fiber surfaces: generalizing isosurfaces to bivariate data. Comput. Graph. Forum **34**, 241–250 (2015)
7. Carr, H., Snoeyink, J.: Representing interpolant topology for contour tree computation. In: H.C. Hege, K. Polthier, G. Scheuermann (eds.) Topology-Based Methods in Visualization II, Mathematics and Visualization, pp. 59–74. Springer, Berlin (2009)
8. Carr, H., Snoeyink, J., Axen, U.: Computing contour trees in all dimensions. Comput. Geom. **24**(2), 75–94 (2003)
9. Carr, H., Snoeyink, J., van de Panne, M.: Simplifying flexible isosurfaces using local geometric measures. In: Proceedings of the Conference on Visualization 2004 (VIS 2004), pp. 497–504. IEEE Computer Society, Washington, DC (2004). https://doi.org/10.1109/VISUAL.2004.96
10. Carr, H., Snoeyink, J., Van De Panne, M.: Flexible isosurfaces: simplifying and displaying scalar topology using the contour tree. Comput. Geom. **43**(1), 42–58 (2010)
11. Carr, H., Tierny, J., Weber, G.: Pathological and test cases for reeb analysis . In: Topology-Based Methods in Visualization 2017 (TopoInVis 2017), pp. 27–28. Tokyo, Japan, February 2017
12. Carr, H.A., Weber, G.H., Sewell, C.M., Ahrens, J.P.: Parallel peak pruning for scalable SMP contour tree computation. In: 2016 IEEE 6th Symposium on Large Data Analysis and Visualization (LDAV), pp. 75–84 (2016). https://doi.org/10.1109/LDAV.2016.7874312
13. Cohen-Steiner, D., Edelsbrunner, H., Harer, J.: Extending persistence using Poincaré and Lefschetz duality. Found. Comput. Math. **9**(1), 79–103 (2009)
14. Connolly, M.L.: Shape complementarity at the hemoglobin $\alpha 1 \beta 1$ subunit interface. Biopolymers **25**(7), 1229–1247 (1986)
15. Dewdney, A.K.: Computer recreations. Sci. Am. **17**, 18–30 (1985)
16. Edelsbrunner, H., Harer, J.: Computational Topology, An Introduction, 1 edn. Americal Methematical Society, Providence (2013)

17. Edelsbrunner, H., Letscher, D., Zomorodian, A.: Topological persistence and simplification. In: Proceedings 41st Annual Symposium on Foundations of Computer Science, pp. 454–463. IEEE (2000)
18. Edelsbrunner, H., Morozov, D., Pascucci, V.: Persistence-sensitive simplification functions on 2-manifolds. In: Proceedings of the Twenty-Second Annual Symposium on Computational Geometry (SCG 2006), pp. 127–134. ACM, New York (2006). https://doi.org/10.1145/1137856.1137878.
19. Gueunet, C., Fortin, P., Jomier, J., Tierny, J.: Contour forests: fast multi-threaded augmented contour trees. In: IEEE Symposium on Large Data Analysis and Visualization. Baltimore (2016). https://hal.archives-ouvertes.fr/hal-01355328
20. Gueunet, C., Fortin, P., Jomier, J., Tierny, J.: Task-based augmented merge trees with fibonacci heaps. In: 2017 IEEE 7th Symposium on Large Data Analysis and Visualization (LDAV), pp. 6–15 (2017). https://doi.org/10.1109/LDAV.2017.8231846
21. Gyulassy, A., Natarajan, V., Pascucci, V., timo Bremer, P., Member, B.H.: A topological approach to simplification of three-dimensional scalar functions. In: IEEE Transactions on Visualization and Computer Graphics, pp. 474–484 (2006)
22. Landge, A.G., et al.: In-situ feature extraction of large scale combustion simulations using segmented merge trees. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC 2014), pp. 1020–1031 (2014). https://doi.org/10.1109/SC.2014.88
23. Li, C., Ovsjanikov, M., Chazal, F.: Persistence-based structural recognition. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 1995–2002 (2014)
24. Maadasamy, S., Doraiswamy, H., Natarajan, V.: A hybrid parallel algorithm for computing and tracking level set topology. In: 2012 19th International Conference on High Performance Computing, pp. 1–10 (2012). https://doi.org/10.1109/HiPC.2012.6507496
25. Matsumoto, Y.: An Introductino to Morse Theory (Translation of Mathematical Monograms), vol. 208, 1st edn. American Mathematical Society, Providence (2002)
26. Morozov, D., Weber, G.: Distributed merge trees. In: Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 93–102 (2013). https://doi.org/10.1145/2517327.2442526
27. Morozov, D., Weber, G.H.: Distributed contour trees. In: Bremer, P.T., Hotz, I., Pascucci, V., Peikert, R. (eds.) Topological Methods in Data Analysis and Visualization III, pp. 89–102. Springer International Publishing, Cham (2014)
28. Pascucci, V., Cole-McLaughlin, K.: Parallel computation of the topology of level sets. Algorithmica **38**(1), 249–268 (2004). https://doi.org/10.1007/s00453-003-1052-3
29. Pascucci, V., Cole-McLaughlin, K., Scorzelli, G.: Multi-resolution computation and presentation of contour trees. In: Proceedings of the Conference on Visualization, Imaging, and Image Processing, pp. 452–290 (2004)
30. Pascucci, V., Tricoche, X., Hagen, H., Tierny, J.: Topological Methods in Data Analysis and Visualization: Theory, Algorithms, and Applications. Springer Science & Business Media, Berlin (2010)
31. Rosen, P., Tu, J., Piegl, L.A.: A hybrid solution to parallel calculation of augmented join trees of scalar fields in any dimension. Comput-Aided Des. Appl. **15**(4), 610–618 (2018). https://doi.org/10.1080/16864360.2017.1419648
32. Shi, Y., Li, J., Toga, A.W.: Persistent Reeb graph matching for fast brain search. In: Wu, G., Zhang, D., Zhou, L, (eds.) Machine Learning in Medical Imaging. MLMI 2014. Lecture Notes in Computer Science, vol. 8679. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-10581-9_38
33. Tarjan, R.E.: Efficiency of a good but not linear set union algorithm. J. ACM **22**, 215–225 (1975)
34. Verovšek, S.K., Mashaghi, A.: Extended topological persistence and contact arrangements in folded linear molecules. Front. Appl. Math. Stat. **2**, 6 (2016)
35. Zomorodian, A.J.: Topology for Computing, 1 edn. Cambridge University Press, Cambridge (2009)