DOI: 10.1111/cgf.12360
Eurographics Conference on Visualization (EuroVis) 2014
H. Carr, P. Rheingans, and H. Schumann
(Guest Editors)

Volume 33 (2014), Number 3

Extended Branch Decomposition Graphs: Structural Comparison of Scalar Data

Himangshu Saikia, Hans-Peter Seidel, Tino Weinkauf
Max Planck Institute for Informatics, Saarbrücken, Germany

Abstract
We present a method to find repeating topological structures in scalar data sets. More precisely, we compare all subtrees of two merge trees against each other – in an efficient manner exploiting redundancy. This provides pair-wise distances between the topological structures defined by sub/superlevel sets, which can be exploited in several applications such as finding similar structures in the same data set, assessing periodic behavior in time-dependent data, and comparing the topology of two different data sets. To do so, we introduce a novel data structure called the extended branch decomposition graph*, which is composed of the branch decompositions of all subtrees of the merge tree. Based on dynamic programming, we provide two highly efficient algorithms for computing and comparing extended branch decomposition graphs. Several applications attest to the utility of our method and its robustness against noise.*

1. Introduction

Structures repeat in both nature and engineering. An example is the symmetric arrangement of the atoms in molecules such as Benzene. A prime example for a periodic process is the combustion in a car engine where the gas concentration in a cylinder exhibits repeating patterns.

Structures repeat with variations. Sometimes these variations are clearly noticeable. For example, the weather patterns over the Atlantic ocean are comparable every summer, but not quite the same. Similarly in engineering, the flow fields around a small and a larger car are not quite the same, but they exhibit significant similarity.

In this paper, we focus on scalar data sets and the topological structures defined by their sub/super-level sets. We enable the user to select such a topological structure – simply by choosing one such set – and search for similar occurrences of it (i) in the same data set, (ii) in different time steps, (iii) in other data sets. We return a list of matches with scores, which can be emphasized in a volume rendering.

Previous research (Section 2) in this direction includes the work of Thomas and Natarajan [TN11, TN13] on detecting symmetry within a scalar data set. In contrast to their work, we lift the restriction to symmetric patterns and enable the comparison between different data sets and time steps.

After recapitulating the theoretical background on topology in Section 3, we provide reasoning towards our approach in Section 4 and present our technical contributions in Section 5:

- We introduce the *extended branch decomposition graph*: a novel data structure that describes the hierarchical decomposition of all subtrees of a join/split tree. We abbreviate it with 'eBDG'.
- We provide a fast algorithm for computing an eBDG. Typical runtimes are in the order of milliseconds.
- We develop an algorithm for comparing two eBDGs. It amounts to comparing *all* subtrees of two join/split trees against each other, but without the redundancy of a naïve 1-to-1 comparison. The algorithm is fast and memory-efficient due to our dynamic programming approach. Typical runtimes are below a second. Subsequently, the user can select any subtree (sub/super-level set) and find topologically similar ones instantly.

The comparison is robust against noise as we show in Section 6 together with a number of other evaluations and applications. Section 7 concludes the paper with a discussion of limitations and future work.

2. Related Work

The largest body of work for finding repeating structures or objects can be found in the computer graphics and computer vision domains, where the task is to look for similar shapes and images, or parts thereof. Many different approaches exist to identify symmetries and similarities; a detailed overview can be found in the state of the art report by Mitra et al. [MPWC13]. Some of these methods use topology [BGSF08]. For example, Hilaga et al. [HSKK01] compare and match shapes using Reeb graphs and Yang et al. [YGW*12] using critical points of the Gaussian curvature.

One possible approach to detecting repeating structures in scalar fields is to extend the methods known from graphics. For example, Kerber et al. [KWKS11] detect partial Euclidean symmetries in volume data by matching crease lines. Hong et al. [HS07] sample a large number of cutting planes through the volume to test for reflective symmetry. These methods are restricted to symmetry and a single data set. Furthermore, they ignore many properties inherent to scalar fields such as the hierarchy of level sets.

Other methods in the visualization domain compare two or more scalar fields with each other, in particular in the context of multi-field analysis. Sauber et al. [STS06] compute pairwise correlations between 3D scalar fields and use a graph structure to navigate the exponentially high number of pairs. Jänicke et al. [JWSK07] compute the statistical complexity for each spatio-temporal location in time-dependent multi-fields, which is a time-dependent scalar field that often correlates with application-specific features. An overview of further methods can be found in [BH07]. In the following, we concentrate on topological approaches. For example, Huettenberger et al. [HHC*13] propose an approach towards multi-field scalar topology by overlaying two scalar fields and assessing their differences using Pareto optimality. Haidacher et al. [HBG11] compare scalar fields from different modalities using an isosurface similarity measure [BM10]. Schneider et al. [SWC*08] extract largest isocontours – as defined by a simplified contour tree – from different 3D data sets and compare their volume overlap as a measure of similarity. Carr and Duke [CD13] propose the concept of the *joint contour net* which expresses the topological relationships between two scalar fields defined over the same domain. It has been applied to the analysis of nuclear fission data sets [DCK*12]. All these methods have in common that they compare two data sets as a whole.

Several methods deal with the comparison of topological structures. Bauer et al. [BGW13] introduce a distance metric for Reeb graphs under which they are stable to small perturbations of the input. The algorithm has exponential complexity. Edelsbrunner et al. [EHMP04] introduce a data structure tracking the evolution of a Reeb graph over time. Knowing the initial structure and incremental changes over time suffices to construct the Reeb graph for any time t. Another topological approach to analyzing time-dependent data is due to [FOTT08]. A close work to ours is due to Beketayev et al. [BYM*13] where two merge trees are compared by means of branch decompositions. They avoid instabilities by considering a large number of these decompositions, which on the other hand leads to very long computation times. Their runtime complexity amounts to $O(N^5)$ for comparing a merge tree with N nodes to itself, whereas our runtime complexity is $O((N \log N)^2)$. Furthermore, their algorithm needs to re-run for every similarity threshold ε, whereas our method computes all scores at once. We pay with the possibility of instabilities (Section 6.1), which can be mitigated by choosing a proper edge weight in the merge tree (Section 6.2).

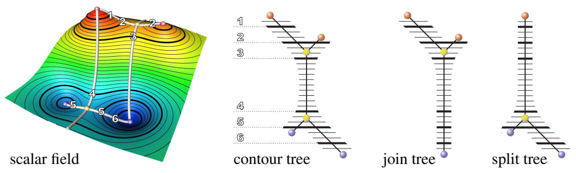


Figure 1: *Contours of a scalar field may appear, join, split, and disappear with changing isovalues. The contour tree represents that behavior. It can be created by merging the join and split tree.*

Tree edit distance is a measure for comparing *labeled* trees [Bil05]. These approaches have exponential computation time. Suboptimal solutions [NRB06] provide significant speedups, but are still far away from being usable in interactive applications.

The closest work to our approach is due to Thomas and Natarajan. They detect symmetric structures in a scalar data set using either the contour tree [TN11] or the extremum graph [TN13]. See Section 6.3 for a discussion.

3. Background

In this section, we discuss the necessary background regarding the topology of scalar fields. Further details can be found in [Car04].

3.1. Contour, Join, Split Trees

Consider a level set of a scalar field $f : \mathbb{R}^n \to \mathbb{R}$ for a given isovalue such as the one denoted with the number “2” in Figure 1. It consists of several connected components, i.e., contiguous parts of the level set, which are called *contours*. When changing the isovalue, new contours may appear, others may disappear, and some contours may join or split. This behavior is recorded in the *contour tree* $T = (\mathcal{V}, \mathcal{E})$. Its nodes $v_i \in \mathcal{V}$ are the minima, maxima, and saddle points of the scalar field. An edge $(v_i, v_j) \in \mathcal{E}$ represents the topologically equivalent contours between v_i and v_j for $f(v_i) > f(v_j)$. Contour trees are unrooted and can be computed by merging two rooted trees: the *join* and *split* tree (often jointly referred to as *merge* trees).

Following the notation of [Car04], the leaves of the join tree are the maxima of the scalar field. Each maximum gives rise to a contour. With decreasing isovalue, they join at saddles until only one contour remains, which will finally collapse at the global minimum, i.e., the root of the join tree. With a slight abuse of notation we write $J = (\mathcal{V}, \mathcal{E})$ for join trees. Reversely, the split tree is rooted at the global maximum and its leaves are the minima of the scalar field.

The edges of a merge tree can be weighted by the function value difference between its nodes. This measure is closely

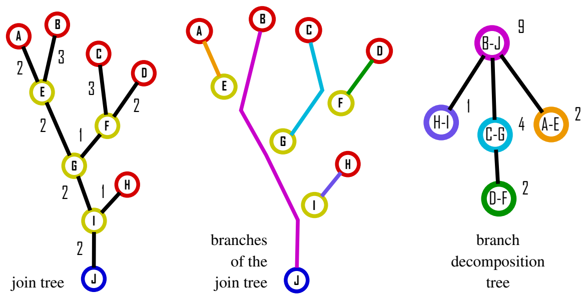


Figure 2: *A join tree can be segmented into a collection of branches according to the weight of the edges. This is a hierarchical segmentation. It is represented as the branch decomposition tree, where each node corresponds to a branch of the join tree.*

related to persistence [ELZ02]. It allows to discriminate dominant and weak features. Other possible weighting measures include the maximum or minimum area/volume of the region represented by an edge. In Section 6.2 we discuss the choice of the weighting function and its influence on the robustness against noise.

3.2. Branch Decomposition Tree

A branch decomposition tree (BDT) is a derivative of a contour/join/split tree. It is usually computed for contour trees [PCMS04], but just as well defined for merge trees – and this is how we will use it in this paper. Furthermore, we restrict the discussion to join trees. The definition for split trees follows in a straightforward fashion.

The BDT aims at representing the contours in a hierarchical manner according to the weight of the edges in the join tree. To this end, the join tree is segmented into non-overlapping branches: a branch consists of one or more edges between a saddle and a maximum. Each branch is represented as a node in the BDT. Figure 2 illustrates this.

Formally, consider a join tree $J = (\mathcal{V}, \mathcal{E})$. We write its branch decomposition tree as $B = (\mathcal{B}, \mathcal{S})$ with the nodes $\mathcal{B} = \{b : b = \{e_i, \dots, e_j\}$ with $e_i, \dots, e_j \in \mathcal{E}\}$ representing the branches, and the edges $\mathcal{S}$ representing their hierarchy.

The computation of the BDT starts with the edge of lowest weight, which gets "plucked" from the join tree and becomes a node in the BDT. This removes a saddle-maximum pair from the join tree and combines two edges. In Figure 2, the edge $(D \rightarrow F)$ is "plucked" off the join tree and becomes a node in the BDT, while the former edges $(C \rightarrow F)$ and $(F \rightarrow G)$ are merged into the branch $(C \rightarrow G)$, whose weight is the sum of the individual edge weights. The computation continues by iteratively removing the branch with the currently lowest weight. Eventually, the root of the BDT is formed by the branch between the global minimum and the global maximum, i.e., $(B \rightarrow J)$ in Figure 2.

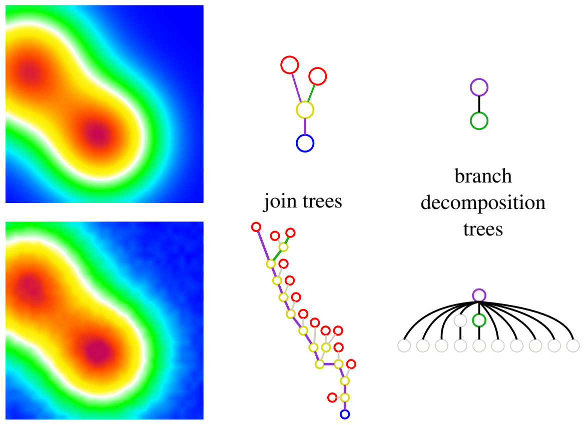


Figure 3: *The upper row shows a simple scalar field with two maxima as well as its corresponding join tree and branch decomposition tree. The lower row shows a version with added noise. Note how difficult it is to relate the two join trees, whereas the branch decomposition trees show a remarkable similarity.*

4. Towards Comparison of Merge Trees

We enable the user to select a topological structure and find similar occurrences of it in the same or another scalar field. A structure is selected by choosing a point in the domain. This defines an isovalue, which corresponds to an edge in the merge tree (see Figure 1). From a topological point of view, the sub/superlevel sets along that edge are equivalent. Hence, the entire edge is included in the selection. Most importantly, the entire subtree is selected, starting at the selected edge and ending at the leaves (minima/maxima).

The task is to find similar occurrences of the selected subtree in a merge tree. Hence, we need an algorithm for computing the similarity of merge trees and parts thereof.

Our approach is based on the extended branch decomposition graph – formally introduced in the next section. In this section, we justify why the merge trees themselves are not well suited for this purpose. Furthermore, we justify why a single branch decomposition tree does not suffice either.

Merge trees represent the nesting of sub/superlevel sets and their edges can be weighted by feature strength (e.g., height difference). However, the notion of feature strength does not manifest in their structure: merge trees are riddled with low-weighted branches throughout their hierarchy. It is not possible to locate the most important edges in a specific part of the tree. Figure 3 illustrates this. A node-to-node comparison is hard, too, since there is no strict order of traversal defined, i.e., the order of the children is undefined. Hence, comparing such trees directly leads to a combinatorial nightmare.

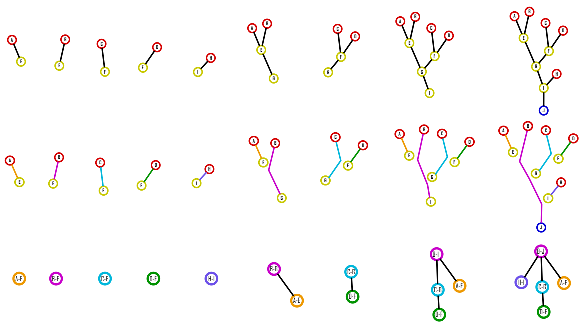


Figure 4: *A forest of all subtrees (top-row) of the join tree shown in Figure 2, along with their branches (middle-row) and the branch decomposition trees (bottom row).*

The branch decomposition tree (BDT) is a better alternative for two reasons:

- A parent node has always a higher feature strength than any of its children. This means that we can compare the most dominant features first by starting a comparison from the root, while noisy structures appear in deeper levels of the BDT. Figure 3 shows how well the BDTs in this example represent the similarity between the two data sets.
- The children in a BDT can be ordered in a meaningful manner: we order them according to the scalar value of their saddles. This facilitates an efficient comparison, since we only need to match the children in a given order, instead of considering all possible permutations.

Our goal is to compare all subtrees of a merge tree. A single BDT does not allow that, since its nodes do not represent all subtrees. Figure 2 shows an example, where the subtrees $(A \rightarrow E \leftarrow B)$ and $(C \rightarrow F \leftarrow D)$ cannot be compared, since they are not represented in a single BDT, which in this case is computed with respect to the root. In order to represent all subtrees, we need just as many BDTs: one BDT for each subtree computed from the root of that subtree.

This leads to a forest of BDTs as shown in Figure 4. It is expensive to keep this forest in memory and to compare all of the trees against each other: let $\mathcal{E}$ be the set of edges of the full merge tree, $\mathcal{E}_i$ be the edges of a subtree and $\mathcal{M}_i$ its extrema. Then it can easily be seen that the memory for storing the forest of BDTs is $\sum_{i=0}^{|\mathcal{E}|}(|\mathcal{E}_i| + |\mathcal{M}_i|)$. The computation of the forest would require $\sum_{i=0}^{|\mathcal{E}|}(O(|\mathcal{E}_i|) + O(|\mathcal{M}_i| \log |\mathcal{M}_i|))$. Naïvely comparing each tree $\mathcal{T}_i$ in this forest against every other tree $\mathcal{T}_j$ in a different forest has a runtime of $\sum_{i=0}^{|\mathcal{E}|}\sum_{i=0}^{|\mathcal{E}|} O(N_i N_j b_i b_j)$, where N_i, N_j denote the sizes of the forests and b_i, b_j the average branching factors in the subtrees. In the next section, we introduce a graph data structure and an efficient comparison algorithm that make this approach feasible.

5. Extended Branch Decomposition Graph

We introduce an efficient graph structure to represent a forest of branch decomposition trees. Each of the BDTs in the forest is computed from a subtree of the merge tree (Section 5.1). We show how to compute the branch decompositions of all subtrees efficiently as part of a memoization process using dynamic programming (Section 5.2). The graph structure is the basis for comparing any subtree of the merge tree with any other subtree. We give an efficient comparison algorithm (Section 5.3). Again, all following explanations will be made for join trees, but work for split trees as well.

5.1. Definition

Let $J = (\mathcal{V}, \mathcal{E})$ denote a join tree. We will now enumerate its subtrees by considering its edges. Each edge $(v_i, v_j) \in \mathcal{E}$ gives rise to one and only one subtree $J_i = (\mathcal{V}_i, \mathcal{E}_i)$ where $\mathcal{V}_i \subseteq \mathcal{V}$ and $\mathcal{E}_i \subseteq \mathcal{E}$. This follows since the edges are directed $v_i \rightarrow v_j$ and there exists one and only one directed path from every node to the global minimum. Hence, for every v_i there exists one and only one v_j where $(v_i, v_j) \in \mathcal{E}$.

For each subtree J_i, we can compute a branch decomposition tree $B_i = (\mathcal{B}_i, \mathcal{S}_i)$. This leads to a forest of branch decomposition trees as shown in Figure 4.

It is crucial to note that the trees in this forest have a substantial overlap. To see this, consider two subtrees J_i and J_j of the join tree J. Furthermore, let J_j be a subtree of J_i. Then, some parts of the hierarchy of B_j will also appear in B_i. This can easily be observed in Figure 4, where the node D-F appears four times in the forest. Furthermore, note that every node in the forest appears also exactly once as a root node. This leads us to a new data structure for representing this forest without redundancy.

The *extended branch decomposition graph* (eBDG) is the union of all branch decomposition trees B_i. We denote it with $B_\cup = (\mathcal{B}_\cup, \mathcal{S}_\cup)$ where $\mathcal{B}_\cup, \mathcal{S}_\cup$ represent the union of the nodes and edges of all B_i. Figure 5 illustrates this.

The following statement holds regarding the number of elements in the eBDG and the join tree:

$$|\mathcal{B}_\cup| = |\mathcal{E}| = |\mathcal{V}| - 1. \tag{1}$$

This follows from two facts:

- As discussed before, every node in the forest of all B_i is also a root node of one particular B_i.
- Every B_i represents a subtree of the join tree, which in turn is represented by the sole edge from the root of the subtree.

In other words, a node of the eBDG uniquely identifies an edge of the join tree, which in turn represents topologically equivalent contours. This is crucial for our applications, since we aim to possess a comparison structure for every contour.

Table 1 shows the eBDG nodes for the contour tree in Figure 2 and exemplifies our indexing scheme for the root

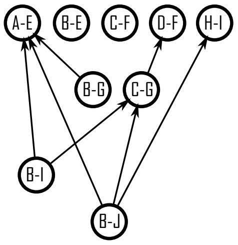


Figure 5: *The extended branch decomposition graph (eBDG) for the join tree in Figure 2 is the union of the BDTs in Figure 4. All BDTs arising from every subtree of the join tree are included in this graph. Every node represents a root branch of a BDT and the rest of the BDT can be traced along the directed edges.*

Index	Root Node	Corresponding Edge in J	Weight	Child Indices	Level
0	A-E	$A \to E$	2	-	0
1	B-E	$B \to E$	3	-	0
2	C-F	$C \to F$	3	-	0
3	D-F	$D \to F$	2	-	0
4	H-I	$H \to I$	1	-	0
5	B-G	$E \to G$	5	0	1
6	C-G	$F \to G$	4	3	1
7	B-I	$G \to I$	7	6, 0	2
8	B-J	$I \to J$	9	4, 6, 0	2

Table 1: *eBDG table for the contour tree in Figure 2.*

nodes of the B_i. The level encodes the hierarchy that a node belongs to. A node without any children is of level-0. A node is level-1, if it has only level-0 children. In general, the level of a node is one higher than the level of its highest-level child.

5.2. Computation

As seen previously, the eBDG avoids redundancy by combining the BDTs of all subtrees of a join tree to a single graph structure. This is possible because subtrees overlap with each other. We exploit this overlapping when computing the eBDG in a bottom-up fashion.

The algorithm starts with the edges of the join tree that contain a maximum. The BDT for such single edges is straightforward: a single node representing the edge. Thus, we initialize the eBDG with these trivial branch decompositions B_i for all maxima $v_i \in \mathcal{V}$:

$$B_i = (e_i, \{\}), \tag{2}$$

where $e_i = (v_i \to v_j) = (v_i, v_j)$ is the unique edge in the join tree starting from v_i. The weights W_i of the nodes B_i and their levels L_i are trivially given by

$$W_i = w_i \qquad L_i = 0 \tag{3}$$

where w_i is the weight of the edge e_i. In Figure 5, this initialization created the nodes in the topmost row, but not any of the edges or other nodes.

The algorithm continues by visiting every saddle of the join tree and updating the eBDG such that it contains the BDT of the respective subtree. The update for each saddle consists of adding a single node and a number of edges to the eBDG. Consider a saddle v_i with $(v_k \to v_i) \in \mathcal{E}$ and all B_k have already been computed. Then we can compute $B_i = (\mathcal{B}_i, \mathcal{S}_i)$ as follows:

$$\ell = \arg\max_x \{W_x | (v_x \to v_i) \in \mathcal{E}\} \tag{4}$$

$$\mathcal{B}_i = e_i \cup \mathcal{B}_\ell \tag{5}$$

$$\mathcal{S}_i = \mathcal{S}_\ell \cup \{(\mathcal{B}_i \to \mathcal{B}_k) | (v_k \to v_i) \in \mathcal{E} \wedge k \neq \ell\} \tag{6}$$

with the weight of the root node of B_i and the level given by

$$W_i = W_\ell + w_i \tag{7}$$

$$L_i = \max(\{1 + L_k | (v_k \to v_i) \in \mathcal{E} \wedge k \neq \ell\} \cup L_\ell). \tag{8}$$

To put this into words: at every saddle v_i, the heaviest branch B_ℓ is identified and combined with the edge e_i to form the root node of B_i. All other branches are assigned as its children. Furthermore, all children of the heaviest branch are also assigned as children of B_i. This takes care of ordering the saddles in a topologically consistent manner, which is required by our comparison algorithm later on.

The above algorithm has space complexity of $O(N)$, where N is the number of nodes in the eBDG. The time complexity is $O(Nd \log d)$, where d is the average branching factor in the join tree, i.e., how many branches merge at a join saddle. Most data sets have only first-order saddles where $d = 2$, so the runtime is linear in N for those cases. This changes when monkey saddles are present (where more than two contours merge at a saddle), but the effect on the runtime is small in all practical cases since d is the *average* branching factor. The complexity is independent of the size of the data, but depends only on the size of the merge tree. Table 2 lists runtimes.

5.3. Comparison

We present a dynamic programming method to compute a comparison score (as a cost function) between any two join trees J^1 and J^2. These trees are compared by means of their eBDGs B^1 and B^2. As a result, we also obtain the comparison scores of all subtrees of J^1 to all subtrees of J^2. This enables us later to do a real-time exploration of similar structures in the data.

The main algorithmic ingredient is to compare an eBDG node to another – including their children. Again, we use a bottom-up approach and memoization. Consider the two eBDGs B^1 and B^2. The cost between two level-0 nodes B_i^1 and B_j^2 is given by the objective function

$$c(B_i^1, B_j^2) = |W_i^1 - W_j^2|. \tag{9}$$

Higher-level nodes have children, which need to be matched up as well such that it leads to the minimum cost. In fact, these children are trees and the actual task is to compare two sequences of trees. The key here is memoization: if the minimum cost of matching a lower-level node is already computed, then we only need to match the immediate children.

Hence, we visit the eBDG nodes in ascending order of their levels and compute their cost as

$$c(B_i^1, B_j^2) = |W_i^1 - W_j^2| + c(F_i^1, F_j^2), \tag{10}$$

where $c(F_i^1, F_j^2)$ is the cost of matching the two sequences of trees below B_i^1 and B_j^2. Since lower-level nodes have been processed first, this is now a simpler matter of matching two sequences of values. Figure 6 illustrates this. Note that such a matching must not have any crossings, since this would lead to topological inconsistencies, where higher-valued saddles would be matched to lower-valued saddles.

Matching two sequences without edge crossings is solved using a divide-and-conquer approach: consider a dissection of the two sequences at any given pair of indices such that optimal matches are not cut. Then the optimal matches within each partial sequence remain the same. Hence, this problem can be solved using dynamic programming. To define the recurrence of our dynamic programming formulation we observe that we have one of three choices at any moment: (i) leave out the first node in the first sequence, (ii) leave out the first node in the second sequence, (iii) match the first two nodes with each other. This leads to the following formulation:

$$c(F_i^1, F_j^2) = d_{0,0}(F_i^1, F_j^2), \tag{11}$$

where $d_{p,q}(F_i^1, F_j^2)$ is the minimal cost of matching the two sequences of trees F_i^1 and F_j^2 at indices p and q, respectively. F_i^1 and F_j^2 are given by:

$$F_i^1 = \{B_{i,0}^1, B_{i,1}^1, B_{i,2}^1, \dots, B_{i,m}^1\} \tag{12}$$

$$F_j^2 = \{B_{j,0}^2, B_{j,1}^2, B_{j,2}^2, \dots, B_{j,n}^2\}. \tag{13}$$

Here, B_i^1 has m children and B_j^2 has n children. Writing $d_{p,q}(\{B_{i,0}^1, B_{i,1}^1, B_{i,2}^1, \dots, B_{i,m}^1\}, \{B_{j,0}^2, B_{j,1}^2, B_{j,2}^2, \dots, B_{j,n}^2\})$ simply as $d_{p,q}$ we have,

$$d_{p,q} = \min \begin{cases} d_{p+1,q} + c(B_{i,p}^1) \\ d_{p,q+1} + c(B_{j,q}^2) \\ d_{p+1,q+1} + c(B_{i,p}^1, B_{j,q}^2) \end{cases} \tag{14}$$

where $p \in \{0, \dots, m\}$ and $q \in \{0, \dots, n\}$. Also $d_{p,n+1} = d_{m+1,q} = 0$. The cost of leaving out a node is equivalent to leaving out all of its children. This is given by:

$$c(B_i) = W_i + c(F_i) \tag{15}$$

$$c(F_i) = \Sigma c(B_k)\ \forall k \in \text{children}(i). \tag{16}$$

Thus, using the memoized scores from previously computed lower-level node matches, an all-to-all score table is obtained. From this, the similarity for any pair of subtrees J_i^1 and J_j^2 can be obtained, including the full join trees J^1 and J^2.

The space complexity of the comparison algorithm is $O(N_1N_2 + b_1b_2)$, where N_1 and N_2 denote the sizes of the two eBDGs, and b_1 and b_2 denote the average branching factors. The time complexity is $O(N_1N_2b_1b_2)$. Consider L_1 and L_2 as the average levels in the two trees. Then we can write the time complexity as

$$O(N_1N_2 \log_{L_1+1}(N_1+1) \log_{L_2+1}(N_2+1)). \tag{17}$$

As an example, for finding similarities within the same data set and for $L = 2$, we have a time complexity of $O((N \log N)^2)$. Again, the complexity is independent of the size of the data, but depends only on the size of the merge tree. Table 2 lists runtimes.

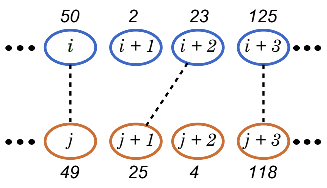


Figure 6: *Illustration of the dynamic programming concept behind matching child nodes.*

Dataset	Vertices	Edges in join tree (after simplification)	eBDG computation	eBDG comparison with itself
Benzene	101^3	23	0.02	0.13
Neghip	64^3	252	0.17	14.93
EMDB-1706	130^3	155	0.12	15.65
EMDB-1603	160^3	38669 (911)	1.89	73.45
EMDB-1603 (z=47)	160^2	1416	1.36	2028.1

Table 2: *Computation times (in milliseconds) for various data sets. All operations were performed on a machine with a 2.66GHz Intel Xeon processor and 12GB main memory.*

6. Evaluation, Comparison and Applications

In the following, we discuss and evaluate further aspects such as the stability of branch decompositions, compare our method to previous work, and describe several applications.

6.1. Stability of Branch Decompositions

Any process of decomposing a merge tree into branches needs to make binary decisions about the vertical (parent-child) and horizontal (siblings) ordering of branches. These binary decisions create instabilities with respect to perturbations. This is also true for the branch decomposition introduced in Section 3.2 and used throughout this paper, which is based on the edge weights in the merge tree.

Figure 7 shows a vertical instability for a 2D scalar field where the edge weights in the merge tree are the function value differences between the nodes. The function values of the two largest maxima B and C are very similar, and a perturbation can lead to a different vertical ordering of their respective branches. Our comparison algorithm gives a score between these two configurations that depends on the weights of the children (the two major branches are similar by design): $2\,|w_{(D\to F)} - w_{(A\to E)}|$.

Figure 8 shows a horizontal instability, where the two saddles E and G are close to each other and swap their position in the merge tree under perturbation. This leads to a

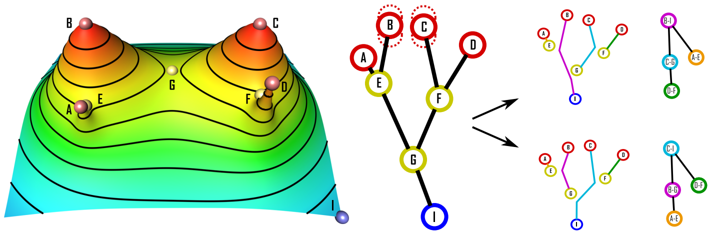


Figure 7: *Vertical instability of a branch decomposition. The extrema B and C fight for the dominance, since the corresponding edge weights are very similar, i.e., small perturbations may favor one over the other.*

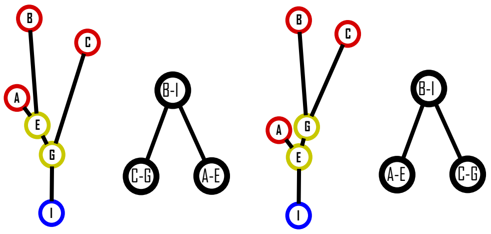


Figure 8: *Horizontal instability of a branch decomposition. Swapping the close saddles E and G leads to a different horizontal ordering in the BDT.*

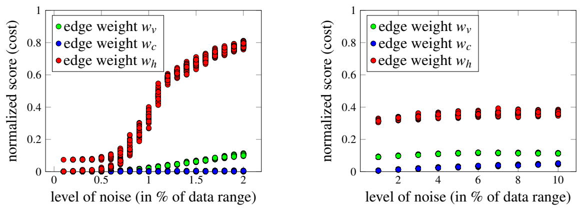


Figure 9: *Perturbation analysis of the data set from Figure 7 (left) and the Neghip data set (right, cf. Figure 12). Shown is the comparison score of the unperturbed data against increasingly noisier versions. The score is normalized by the total weight of the unperturbed BDT. At every noise level, 50 differently perturbed samples are shown. They show different patterns depending on the choice of edge weights for the merge tree.*

different horizontal ordering in the BDT. Our comparison algorithm gives a score between these two configurations: $2\min[|w_{(C\to G)} - w_{(A\to E)}|,\ w_{(C\to G)},\ w_{(A\to E)}]$.

These instabilities induce undesired scores. In fact, when comparing a function to a perturbed version of itself, then the returned score should correlate to the range of the perturbation. Figure 9 shows an experiment regarding this using the data set from Figure 7. We compare the unperturbed data with a large number of perturbed versions. The comparison scores are shown as red dots. As expected, they form two clusters for low levels of noise, precisely because of the vertical instability described above. This effect is dwarfed at higher levels of noise by other topological changes.

Beketayev et al. [BYM*13] avoid these instabilities by considering branch decompositions with permuted branches, thereby explicitly allowing that parents have lower weights than their children. Since the number of these permutations grows exponentially with the number of extrema [BYM*13], this leads quickly to infeasible computation times. Therefore, we propose to use only the default branch decomposition and study the actual effect of these instabilities in real data sets in the next section.

6.2. Choice of Edge Weights and Noise Robustness

The choice of the edge weights for the merge tree influences the occurrence of instabilities. This is easy to see: consider the same weight for all edges, then we have a large number of instabilities. Consider distinctively different weights for all edges and there would be no instabilities. We discuss the following weights for an edge $(a \to b)$:

- Height (function value) difference: $w_h = |f(a) - f(b)|$
- Volume (area) covered by the contour of $(a \to b)$: w_v
- A combination of the above: $w_c = w_h \cdot w_v$

These measures are normalized before use, i.e., w_h is normalized by the data range, and w_v is normalized by the volume of the data set. Hence, all weights are in the range $[0,1]$.

Figure 9 shows the previously described perturbation analysis for the three choices of edge weights. Further plots can be found in the supplemental material. They show how the comparison score behaves under smaller and larger perturbations. Erratic behavior of the score indicates instabilities and would render it useless for any application. As it turns out, the score shows a relatively large variance for the the weights w_h and w_v, but it develops smoothly and with low variance for the combined weight w_c. Hence, we deem w_c to be a good choice. Note that the scores in these plots *are comparable*, because we normalize them to the total weight of their respective unperturbed BDTs. In other words, if half of the respective BDT changes, then it shows in these plots as a cost of 0.5.

6.3. Comparison to Previous Work on Self-Similarity

One of the applications (next section) of our method is self-similarity, i.e., finding similar structures within the same data set. Symmetry can be seen as a special form of self-similarity. Hence, we deem it necessary to compare our approach in more detail to the approaches of Thomas and Natarajan, who find symmetry in a scalar field using the branch decomposition tree [TN11], or using extremum graphs [TN13].

As discussed in Section 4, the BDT does not address all subtrees of a merge or contour tree. Hence, [TN11] is not able

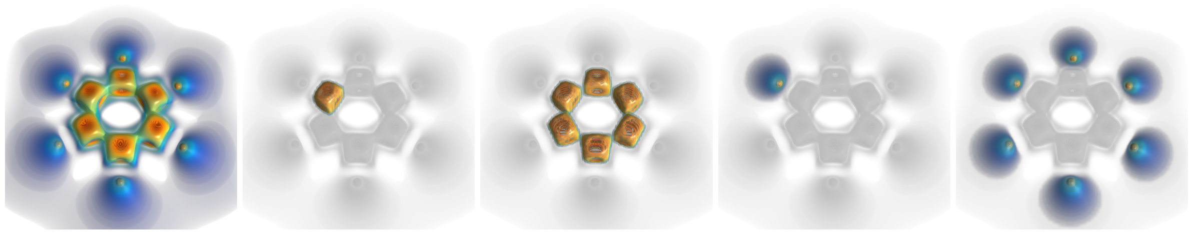

(a) Full volume rendering. (b) Selected ROI. (c) Similar structures. (d) Different ROI. (e) Structures similar to (d).

Figure 11: *Self-Similarity pertaining to different regions in the electrostatic field of the Benzene molecule.*

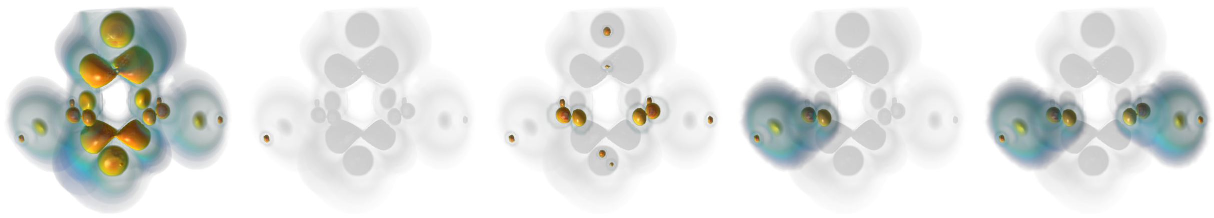

(a) Full volume rendering. (b) Small selected ROI. (c) Similar structures. (d) Expanded ROI. (e) Similar structures.

Figure 12: *Interactive expansion of search regions in the Neghip data set. A small ROI is expanded to the next higher level in the join tree and the corresponding matches are instantly updated. Notice that complicated self-similar structures are found.*

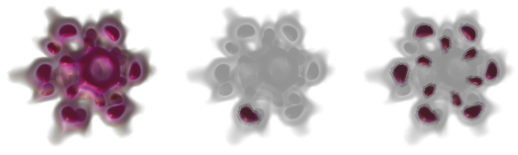

(a) Full volume rendering. (b) Selected super-seed. (c) All other super-seeds.

Figure 10: *Automatic super-seed selection from one super-seed for the method in [TN13]. EMDB-1706 data set.*

to handle all topological structures of the data set as they are defined by the contour tree. Our method is able to do that for merge trees, since the eBDG subsumes all these structures. On the other hand, [TN11] works with the contour tree which holds more information than a single merge tree.

Thomas and Natarajan [TN11] group similar branches of the branch decomposition tree and displays all identified groups using different transfer functions. This is a very useful feature as it allows to identify structures without user interaction. In our setting, we could implement that by clustering the nodes in the eBDG. This could for example be possible by solving the clique problem. We leave that to future work.

The extremum graph approach [TN13] aims at incorporating more geometric information into the computation. This is very useful, since symmetry is a geometric feature. As we will describe in the next section, we incorporate geometric information as well, but with a different approach: we allow to encode volume measures in the edge weights instead of just using the height difference.

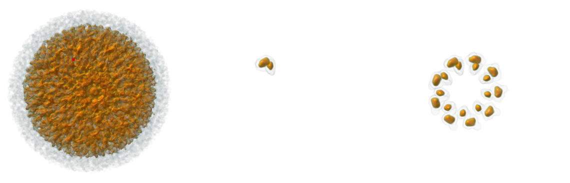
The extremum graph approach [TN13] has one disadvantage: one has to manually define so-called super-seeds (locations in the domain) to actually find and distinguish all symmetric structures. See Figure 11 in [TN13]. As we show in Figure 10 for the same data set, our method can actually help with the creation of a set of super-seeds. The user can select one region of interest and our method returns all similar occurrences, which can then be used as super-seeds in [TN13].

6.4. Applications

The computation and comparison of eBDGs is usually in the order of milliseconds as listed in Table 2. Note that we do *not* simplify the merge trees for most data sets, but this may become necessary for topologically very rich data sets such as the EMDB-1603 from Figure 13. A plot of simplification threshold vs. computation times can be found in the supplemental material.

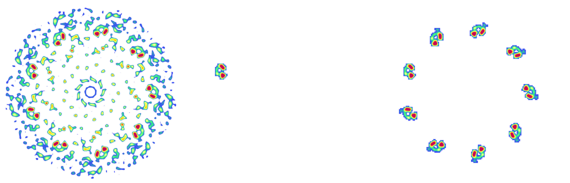
6.4.1. Self-Similarity in a Data Set

In this application, we want to find topologically similar structures in the same data set. To do so, we just need to compare the eBDG to itself. Since the comparison scores



(a) Full volume. (b) Selected ROI. (c) Similar structures.

Figure 13: *The EMDB-1603 data set has ≈39k edges in the join tree. We simplified it to ≈900 edges.*



(a) Full slice. (b) Selected ROI. (c) Similar structures.

Figure 14: *Self-similarity in a slice of EMDB-1603.*

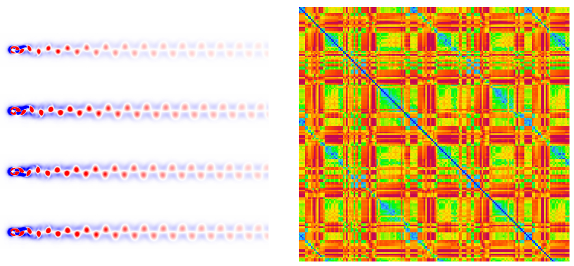
for all subtrees against all other subtrees are precomputed, this allows for a *real-time exploration* of the scalar field for self-similar structures. The user defines a region of interest (ROI) using a point in the domain and a range in the merge tree. The system answers by highlighting similar regions.

Figures 11-14 show examples of self-similarities for different data sets. Note how the results are not restricted by scaling or other Euclidean transformations. Since there are scores involved, we could display similarities at different thresholds, for example from very similar to quite distinct. The real-time exploration of the data allows us to shift our ROI to other interesting regions (Figure 11), expand it to higher-level nodes (Figure 12), and estimate self-similarities in large and noisy data sets using topological simplification (Figure 13).

Some volume data sets consist of very interesting symmetric patterns which are not directly evident when visualizing the full data set in 3D. Occlusion is the main problem. In these cases, a certain slice of the data set can be isolated and symmetric patterns can be explored within this slice. Figure 14 illustrates such an example.

6.4.2. Periodicity in Time-Dependent Data

Whether a time dependent data set exhibits periodic patterns is of importance in fluid dynamics [GTP*07] and flow analysis [STW*06]. Comparing two contour trees from two different times steps can tell us if similar structures manifest themselves at different time intervals. Figure 15 shows a 2D time-dependent flow behind a cylinder. It exhibits periodic vortex shedding [ZFN*95]. We compare the full join tree of the first time step to the join trees from all other 1000 time steps. The resulting similarity scores reveal the periodic pattern in the data. Please observe the diagonal regions bounded by bluish lines. From this, we can approximate a period length of ≈76 time steps. In Figure 15(a) we see that the time slices corresponding to $T = 0$ and $T = 75$ are structurally very similar which supports our approximated period.



(a) Time steps from top: $T = 0$, $T = 10$, $T = 35$ and $T = 75$. (b) Score matrix, from blue (similar) to red (different).

Figure 15: *Periodicity analysis in the 2D time-dependent flow behind a cylinder. The scores are obtained by comparing the full join trees of every time step against each other: for* 1000 *time slices, a* 1000×1000 *symmetric matrix is obtained.*

7. Conclusions

We introduced a novel data structure: the *extended branch decomposition graph*. It is the union of all branch decompositions of all subtrees of a merge tree. In contrast to the forest of all such branch decomposition trees, the eBDG is free of redundancy. We provided an algorithm for computing eBDGs and an algorithm for comparing them. As we show with our complexity analysis and in our evaluations, both algorithms are fast and memory-efficient. We compared our method against previous work. We have shown several applications that benefit from using eBDGs to find topologically similar structures in data sets.

Several possibilities are open for future work. As already mentioned, it would be interesting to find self-similar patterns automatically without choosing a region. This requires to cluster the eBDG nodes, thereby solving the clique problem.

Another open question is how to define the eBDG for contour trees. The main issue is the dynamic programming regarding matching child nodes (see Figure 6): currently, we just need to find an optimal matching between the two sequences, since all children are of the same type. But if the children can be of different type (join or split), we have to consider significantly more possibilities at this stage of the comparison algorithm. We refrained from that for this paper, but plan to work on it in the future. In fact, we found that in many applications, one tree (join or split) suffices to select interesting structures.

References

[BGSF08] BIASOTTI S., GIORGI D., SPAGNUOLO M., FALCIDIENO B.: Reeb graphs for shape analysis and applications. *Theor. Comput. Sci. 392*, 1-3 (Feb. 2008), 5–22. 1

[BGW13] BAUER U., GE X., WANG Y.: Measuring distance between reeb graphs. *CoRR abs/1307.2839* (2013). 2

[BH07] BÜRGER R., HAUSER H.: Visualization of multi-variate scientific data. *EuroGraphics State of the Art Reports (STARs)* (2007), 117–134. 2

[Bil05] BILLE P.: A survey on tree edit distance and related problems. *Theor. Comput. Sci 337* (2005), 217–239. 2

[BM10] BRUCKNER S., MÖLLER T.: Isosurface similarity maps. *Computer Graphics Forum 29*, 3 (2010), 773–782. 2

[BYM*13] BEKETAYEV K., YELIUSSIZOV D., MOROZOV D., WEBER G. H., HAMANN B.: Measuring the distance between merge trees. In *TopoInVis* (2013). 2, 7

[Car04] CARR H.: *Topological Manipulation of Isosurfaces*. PhD thesis, The University of British Columbia, 2004. 2

[CD13] CARR H., DUKE D.: Joint contour nets: Computation and properties. In *Proc. PacificVis* (2013), pp. 161–168. 2

[DCK*12] DUKE D., CARR H., KNOLL A., SCHUNCK N., NAM H. A., STASZCZAK A.: Visualizing nuclear scission through a multifield extension of topological analysis. *IEEE TVCG 18*, 12 (2012), 2033–2040. 2

[EHMP04] EDELSBRUNNER H., HARER J., MASCARENHAS A., PASCUCCI V.: Time-varying reeb graphs for continuous space-time data. In *Proc. Symposium on Computational Geometry* (2004), ACM, pp. 366–372. 2

[ELZ02] EDELSBRUNNER H., LETSCHER D., ZOMORODIAN A.: Topological persistence and simplification. *Discrete and Computational Geometry 28*, 4 (2002), 511 – 533. 3

[FOTT08] FUJISHIRO I., OTSUKA R., TAKAHASHI S., TAKESHIMA Y.: T-map: A topological approach to visual exploration of time-varying volume data. In *High-Performance Computing* (2008), Springer, pp. 176–190. 2

[GTP*07] GÜNTHER B., THIELE F., PETZ R., NITSCHE W., SAHNER J., WEINKAUF T., HEGE H.-C.: Control of separation on the flap of a three-element high-lift configuration. In *45th AIAA Aerospace Sciences Meeting and Exhibit* (Reno, U.S.A., January 2007). AIAA-2007-265. 9

[HBG11] HAIDACHER M., BRUCKNER S., GRÖLLER M. E.: Volume analysis using multimodal surface similarity. *IEEE Transactions on Visualization and Computer Graphics 17*, 12 (Oct. 2011), 1969–1978. 2

[HHC*13] HUETTENBERGER L., HEINE C., CARR H., SCHEUERMANN G., GARTH C.: Towards multifield scalar topology based on pareto optimality. *Computer Graphics Forum 32*, 3pt3 (2013), 341–350. 2

[HS07] HONG Y., SHEN H.-W.: Parallel reflective symmetry transformation for volume data. In *Proc. Eurographics conference on Parallel Graphics and Visualization* (2007), pp. 77–83. 2

[HSKK01] HILAGA M., SHINAGAWA Y., KOHMURA T., KUNII T.: Topology matching for fully automatic similarity estimation of 3D shapes. In *Proc. SIGGRAPH* (2001), pp. 203–212. 1

[JWSK07] JÄNICKE H., WIEBEL A., SCHEUERMANN G., KOLLMANN W.: Multifield visualization using local statistical complexity. *IEEE TVCG 13*, 6 (2007), 1384–1391. 2

[KWKS11] KERBER J., WAND M., KRÜGER J., SEIDEL H.-P.: Partial symmetry detection in volume data. In *Vision, Modeling, and Visualization* (2011), pp. 41–48. 2

[MPWC13] MITRA N. J., PAULY M., WAND M., CEYLAN D.: Symmetry in 3D geometry: Extraction and applications. In *Computer Graphics Forum* (2013), Wiley Online Library. 1

[NRB06] NEUHAUS M., RIESEN K., BUNKE H.: Fast suboptimal algorithms for the computation of graph edit distance. In *Proceedings of the 2006 Joint IAPR International Conference on Structural, Syntactic, and Statistical Pattern Recognition* (Berlin, Heidelberg, 2006), SSPR'06/SPR'06, Springer-Verlag, pp. 163–172. 2

[PCMS04] PASCUCCI V., COLE-MCLAUGHLIN K., SCORZELLI G.: Multi-resolution computation and presentation of contour trees. In *Proc. IASTED Conference on Visualization, Imaging, and Image Processing* (2004), Citeseer, pp. 452–290. 3

[STS06] SAUBER N., THEISEL H., SEIDEL H.-P.: Multifield-graphs: An approach to visualizing correlations in multifield scalar data. *IEEE TVCG 12*, 5 (2006), 917–924. 2

[STW*06] SHI K., THEISEL H., WEINKAUF T., HAUSER H., HEGE H.-C., SEIDEL H.-P.: Path line oriented topology for periodic 2D time-dependent vector fields. In *Proc. Eurographics / IEEE VGTC Symposium on Visualization (EuroVis '06)* (Lisbon, Portugal, May 2006), pp. 139–146. 9

[SWC*08] SCHNEIDER D., WIEBEL A., CARR H., HLAWITSCHKA M., SCHEUERMANN G.: Interactive comparison of scalar fields based on largest contours with applications to flow visualization. *Visualization and Computer Graphics, IEEE Transactions on 14*, 6 (2008), 1475–1482. 2

[TN11] THOMAS D. M., NATARAJAN V.: Symmetry in scalar field topology. *IEEE TVCG 17*, 12 (2011), 2035–2044. 1, 2, 7, 8

[TN13] THOMAS D. M., NATARAJAN V.: Detecting symmetry in scalar fields using augmented extremum graphs. *IEEE TVCG 19*, 12 (2013), 2663–2672. 1, 2, 7, 8

[YGW*12] YANG Y., GÜNTHER D., WUHRER S., BRUNTON A., IVRISSIMTZIS I., SEIDEL H.-P., WEINKAUF T.: Correspondences of persistent feature points on near-isometric surfaces. In *Proceedings of the Fifth Workshop on Non-Rigid Shape Analysis and Deformable Image Alignment (NORDIA) in Proceedings of ECCV 2012 and its Workshops* (Florence, Italy, October 2012), pp. 102–112. 1

[ZFN*95] ZHANG H.-Q., FEY U., NOACK B., KÖNIG M., ECKELMANN H.: On the transition of the cylinder wake. *Phys. Fluids 7*, 4 (1995), 779–795. 9