Fast Topology-based Feature Tracking using a Directed Acyclic Graph

Himangshu Saikia and Tino Weinkauf

Abstract We present a method for tracking regions defined by Merge trees in time-dependent scalar fields. We build upon a recently published method that computes a directed acyclic graph (DAG) from local tracking information such as overlap and similarity, and tracks a single region by solving a *shortest path* problem in the DAG. However, the existing method is only able to track one selected region. Tracking *all* regions is not straightforward: the naïve version, tracking regions one by one, is very slow. We present a fast method here that tracks all regions at once. We showcase speedups of up to two orders of magnitude.

1 Introduction

We are concerned with the tracking of regions defined by merge trees. In [14], we devised a method that tracks the superlevel or sublevel sets of a scalar field as defined by the subtrees of the merge tree. However, once these regions have been extracted in each time step, we neglect their origin and record tracking information such as overlap and histogram similarity in a directed acyclic graph (DAG). Its nodes are the regions. Overlapping and similar regions in consecutive time steps are connected by an edge, weighted by the amount of overlap and similarity. We solve a *shortest path* problem to track a region over time. This global approach to tracking prevents the issue with only local decisions as shown in Figure 1.

In [14], we present, among other things, a method for tracking a single region using the DAG.This is done by computing shortest paths to all reachable sources and sinks from a given node and combining those two paths. This however, is not how one might define a shortest path via a node. In this paper, we define a shortest

Himangshu Saikia
KTH Royal Institute of Technology, Stockholm, Sweden, e-mail: saikia@kth.se

Tino Weinkauf
KTH Royal Institute of Technology, Stockholm, Sweden e-mail: weinkauf@kth.se

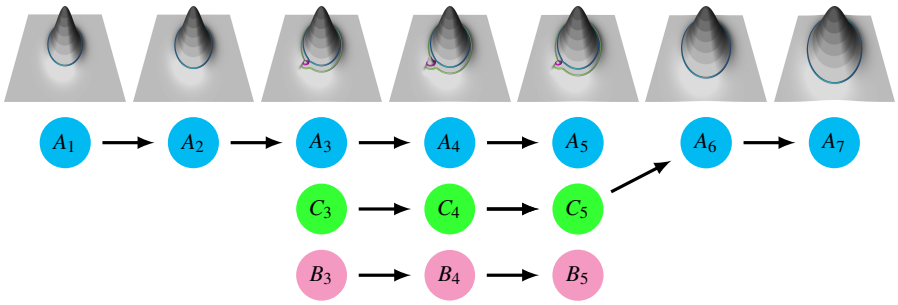


Fig. 1 Tracking regions solely based on local decisions leads to broken tracks. In this simple example, a small fluctuation between time steps t_3 and t_5 causes the creation of a region C that has significant overlap and similarity with region A. Assigning the locally best match neglects that there can be more than one suitable track between two time steps (e.g., between t_5 and t_6), and causes tracks to break. A graph structure as illustrated in Figures 3a and 3b helps circumvent this problem.

path as the path with the least objective function value out of all paths starting at a source, going through the given node and ending at a sink. An objective function can be any function which assigns a score to a path based on how well it represents the evolution of a particular feature along that path. Using this definition of a shortest path, the previous method of combining backward and forward shortest paths may not work.

In this work, we extend the previous work and present a non-trivial solution to tracking all regions from all time steps, i.e., a method for extracting all feature tracks. The trivial solution is to iterate over all nodes of the DAG and execute the single region tracking algorithm from [14]. However, we will show in this paper how this leads to very long running times. Our approach is up to two orders of magnitude faster. Our method employs a shortest path algorithm but is quite different from the standard Djikstra's algorithm or Floyd-Warshall's algorithm to compute all pairs shortest paths. Our DAG being structured in a way that only temporal edges - edges between nodes of two successive time steps - exist, facilitates better runtime bounds than standard algorithms.

2 Related Work

The sheer size of time-dependent data sets often necessitates a data reduction step before an efficient analysis can take place. It is therefore a common approach to extract and track features.

Many methods track topological structures. Tricoche et al. [23] track critical points and other topological structures in 2D flows by exploiting the linearity of the underlying triangle grid. Garth et al. [4] extend this to 3D flows. Theisel and Weinkauf [19, 27] developed feature flow fields as a general concept to track many

different features independent of the underlying grid. Reinighaus et al. [11] extended this idea to the discrete setting.

In the area of time-dependent scalar fields several methods exist to track and visualize topological changes over time. Samtaney et al. [15] provides one of the first algorithms to track regions in 3D data over time using overlap. Kettner et al. [7] presents a geometric basis for visualization of time-varying volume data of one or several variables. Szymczak [17] provides a method to query different attributes of contours as they merge and split over a certain time interval. Sohn and Bajaj [16] presents a tracking graph of contour components of the contour tree and use it to detect significant topological and geometric evolutions. Bremer et al. [2] provide an interactive framework to visualize temporal evolution of topological features.

Other methods for tracking the evolution of merge trees such as the method due to Oesterling [8] track changes to the hierarchy of the tree. This comes at the price of a very high computation time. Its runtime complexity is polynomial in the data size, more precisely, it is $O(n^3)$ with n being the number of voxels. However, the method tracks the unaugmented (full) merge tree instead of just critical points or super-arcs.

Vortex structures are another important class of features that can be tracked in time-dependent flows. Reinders et al. [10] track them by solving a correspondence problem between time steps based on the attributes of the vortices. Bauer and Peikert [1] and Theisel et al. [18] provide different methods for tracking vortices defined by swirling stream lines. This notion was extended later to include swirling path lines [26], swirling streak and time lines [25], swirling trajectories of inertial particles [6] and rotation invariant vortices [5].

Pattern matching has been originally developed in the computer vision community. A number of visualization methods have been inspired by that. Examples are pattern matching methods for vector fields based on moment invariants as proposed by Bujack et al. [3], or pattern matching for multi-fields based on the SIFT descriptor as proposed by Wang et al. [24].

A similar line of research, but technologically rather different, is the analysis of structural similarity in scalar fields, which gained popularity recently. Thomas and Natarajan detect symmetric structures in a scalar field using either the contour tree [21], the extremum graph [22], or by clustering contours [20]. Saikia et al. compared merge trees by means of their branch decompositions [12] or by means of histogram over parts of the merge tree [13]. Our method outputs a set of best tracks of topologically segmented structures in a spatio-temporal setting, and enables an all-to-all temporal pattern matching scheme using techniques like dynamic time warping.

3 Method

In the following, we will first briefly recapitulate the tracking method for single regions of [14], and then present our new and fast approach for tracking all regions.

3.1 Tracking Merge Tree Regions using a Directed Acyclic Graph

Given is a time-dependent scalar field. It could have any number of spatial dimensions, our implementation supports 2D and 3D. A merge tree is computed from each time step independently. After an optional simplification, all subtrees (as defined in [12]) are converted into a set of nodes to be used within the Directed Acyclic Graph (DAG). They represent the components of the superlevel or sublevel sets of the scalar field and are continuous regions in the domain.

All overlapping nodes from consecutive time steps are connected via edges in the DAG. Their weights represent local tracking information in the sense that a lower edge weight indicates a higher likelihood for the two connected regions to be part of the same track. We use a linear combination of *volume overlap* and a *histogram difference* to compute these weights.The *volume overlap* distance d_o between two non-empty regions $\mathcal{S}_a$ and $\mathcal{S}_b$ is determined from the number of voxels they have in common and the total number of voxels covered by both regions:

$$d_o(\mathcal{S}_a, \mathcal{S}_b) = 1 - \frac{|\mathcal{S}_a \cap \mathcal{S}_b|}{|\mathcal{S}_a \cup \mathcal{S}_b|} \quad . \tag{1}$$

The Chi-Squared histogram distance (see e.g. [9]) between two regions is defined as

$$d_s(\mathcal{S}_a, \mathcal{S}_b) = \chi^2(\mathbf{h}_a, \mathbf{h}_b) = \frac{1}{2} \sum_i \frac{(h_{a,i} - h_{b,i})^2}{h_{a,i} + h_{b,i}} \ , \tag{2}$$

where $h_{a,i}$ and $h_{b,i}$ denote the bins of the histograms $\mathbf{h}_a$ and $\mathbf{h}_b$, respectively. Here the histograms represent the number of vertices encapsulated by a region as described in [13].

Our combined distance measure for an edge is given by $d = \lambda d_s + (1-\lambda) d_o$ where $\lambda \in [0,1]$ is a tunable parameter. It is now possible to use this DAG for the next step as is, or it can be further thresholded to weed out extremely large weighted edges (For instance in Figure 3a the edges between the green and pink nodes are removed).

We track a region by solving a shortest path problem with Dijkstra's algorithm on the DAG. The method in [14] does this for one region at a time. From the selected region, a shortest path is found to a source in an earlier time step, and another shortest path is found to a sink in a later time step. Combining these paths yields the track of the region. We describe a path to be a set of successive directed edges in the graph. Since there exists only a single directed edge between any two nodes in the graph, a path can also be described by all successive nodes that connect these edges. Source and sink refer in this context to nodes that have no incoming or outgoing edges, respectively. We discuss this in more detail in the next section.

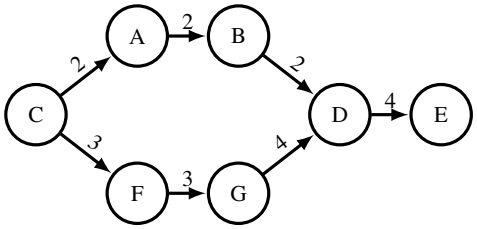


Fig. 2 Illustration of an objective function that does not satisfy the condition expressed in Equation 4. If f signifies the standard deviation of the weights along a path, $f(CABD) < f(CFGD)$ but $f(CABD \cup DE) > f(CFGD \cup DE)$.

3.2 Objective Function and its Validity with Dijkstra's Algorithm

The classic Dijkstra algorithm finds the shortest path by summing up the edge weights along the path. Applying this directly to our setting would yield unsuitable tracks: instead of following a long path with many likely edges, the tracking would rather choose an unlikely edge to an immediate sink.

Hence, we use a measure assessing the average edge weight along a path. The goal is to find the path through a given node that has the smallest normalized squared sum of edge weights d_i:

$$f(\mathcal{P}) = \sqrt{\frac{\sum_{i \in \mathcal{P}} d_i^2}{|\mathcal{P}|}} \tag{3}$$

The purpose of this section is to demonstrate that the Dijkstra algorithm can be used to solve for this objective. To do so, let us define an objective function f which assigns a non-negative score to a path satisfying the following condition:

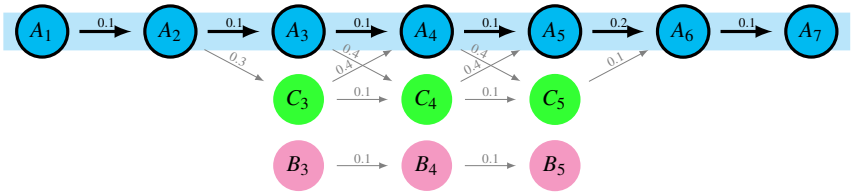
Condition 1: Consider two paths $\mathcal{P}_1$ and $\mathcal{P}_2$ with $f(\mathcal{P}_1) \leq f(\mathcal{P}_2)$. We require the objective function to maintain this relationship after adding an edge $\mathbf{e}$:

$$f(\mathcal{P}_1 \cup \mathbf{e}) \leq f(\mathcal{P}_2 \cup \mathbf{e}) \tag{4}$$

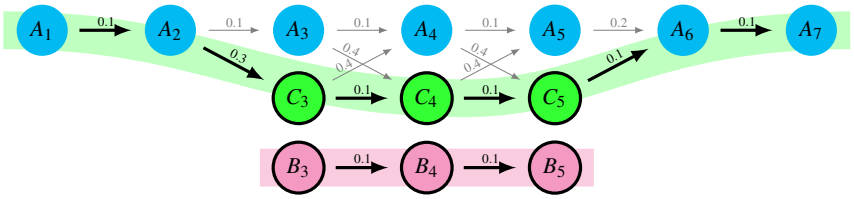
Dijkstra's algorithm can only be used to solve for an objective if this condition is fulfilled, since it allows to incrementally build a solution, which is the essential cornerstone of Dijkstra's algorithm.

The objective function used in the classic Dijkstra's shortest path algorithm is the sum of weights of all edges in a path $\sum_{i \in \mathcal{P}} d_i$. This function trivially satisfies the above condition. A non-satisfying objective function is the standard deviation of the weights as shown in Figure 2. Thus we can see that not all objective functions that determine the quality of a path can be used with Dijkstra's algorithm.

Regarding the objective function (3) we note that it can be solved with Dijkstra's algorithm if $|\mathcal{P}_1| = |\mathcal{P}_2|$ holds, i.e., the two paths are of equal length. This keeps



(a) Dijkstra's algorithm to find the shortest path through the DAG represents the track of this region. Several regions may have the same source and sink, and result in the same shortest path. In this example, the shortest path starting from source A_1 and sink A_7, is common to all the nodes in bold outline. The path is shown as a blue band.



(b) The shortest path through the green bold outlined nodes also has the same source-sink pair (A_1, A_7) as in Figure 3a. It is given by the green band. The shortest path through the red bold outlined nodes is given by the magenta band.

Fig. 3 Several nodes in the graph can have the same shortest path. Hence, running Djikstra's algorithm for every node independently will be expensive and redundant.

the denominator of (3) equal, and the numerators are just a sum of values consistent with Condition 1. The condition $|\mathcal{P}_1| = |\mathcal{P}_2|$ always holds true in our setting, since edges connect two consecutive time steps only and we start Dijkstra's algorithm at a particular source, which keeps all considered paths at equal length. Hence, Dijkstra's algorithm can be used to solve (3).

3.3 Algorithm for Finding All Paths

Tracking of a single node in the DAG is done by finding the shortest path $\mathcal{P}_{min}$ through that node from any source to any sink of the DAG. It may be that the shortest path through other nodes coincides with $\mathcal{P}_{min}$. This is illustrated in Figure 3. Hence, to find the shortest paths through all nodes, running a naïve Dijkstra for every node independently will be expensive and redundant.

Instead, we run Dijkstra's algorithm for every source and sink (in a joint fashion in two passes, see below), record the gathered information at every node, and stitch this information together to obtain the shortest path for every node.

To facilitate this, we define a function to incrementally compute the objective function in (3). We denote this new function by the symbol $\oplus$ and call it the *in-*

cremental path operator. The incremental path operator takes as input the objective value for path $\mathcal{P}$ and a connecting node $\mathbf{n}$ and computes the global measure for path $\mathcal{P} \cup \mathbf{n}$. If the weight of the connecting edge between $\mathcal{P}$ and $\mathbf{n}$ is given by d, $\oplus$ is defined as follows

$$f(\mathcal{P} \cup \mathbf{n}) = f(\mathcal{P}) \oplus d = \sqrt{\frac{f(\mathcal{P})^2 \cdot |\mathcal{P}| + d^2}{|\mathcal{P}| + 1}} \tag{5}$$

Furthermore, all nodes are topologically sorted. That is, for a node $n_{p,i}$ at timestep $t = p$ and another node $n_{q,j}$ at timestep $t = q$, node $n_{p,i}$ occurs before node $n_{q,j}$ in the sorted order if $p < q$.

Our algorithm works as follows. We make two passes through this list of sorted nodes. One in the sorted order (past time steps to future time steps) and one in the reverse sorted order. During the first pass, at every node, the best path from every reachable source to that given node is recorded. This is done by checking all incoming edges to that node and incrementally calculating the best path from all incoming edges from a single source. This becomes possible because all nodes connected to the incoming edges have already been processed earlier (they live at the previous time step). Consider a node n_i with some incoming edges as illustrated in Figure 4. The best score from any given source to n_i is calculated using:

$$f(\mathcal{P}_{src \to i}) = \min_{n_j \in \mathbf{I}_i \wedge bestSource_j = src} (f(\mathcal{P}_{src \to j}) \oplus d_{j,i}). \tag{6}$$

Algorithm 1 shows the pseudo-code for the first pass described above. The second pass is equivalent to the first, but operates on the DAG with edges reversed. We record the best path to every reachable sink now. This is done by checking for outgoing edges and the sinks that they lead to. The best score to any given sink from n_i is calculated using:

$$f(\mathcal{P}_{i \to sink}) = \min_{n_j \in \mathbf{O}_i \wedge bestSink_j = sink} (f(\mathcal{P}_{j \to sink}) \oplus d_{i,j}) \tag{7}$$

Let the set of all reachable sources to node $\mathbf{n}_i$ be $\mathbf{S}_i$ and the set of all reachable sinks be called $\mathbf{K}_i$. After the two passes are complete, the combined best path for every node is calculated by choosing the paths $(\mathcal{P}^- \in \mathbf{S}_i, \mathcal{P}^+ \in \mathbf{K}_i)$ from the source-sink pair which minimizes the objective function on the combined path $\mathcal{P}^- \cup \mathcal{P}^+$ as follows:

$$\mathcal{P}_{\min} = \operatorname*{arg\,min}_{\mathcal{P}^- \in \mathbf{S}_i, \mathcal{P}^+ \in \mathbf{K}_i} \sqrt{\frac{|\mathcal{P}^-| \cdot f(\mathcal{P}^-)^2 + |\mathcal{P}^+| \cdot f(\mathcal{P}^+)^2}{|\mathcal{P}^-| + |\mathcal{P}^+|}}. \tag{8}$$

Algorithm 2 shows the pseudo-code to obtain all best paths. It can be observed that, if for any given node $\mathbf{n}_i$, the best source-sink pair is given by $(\mathbf{s}_i, \mathbf{k}_i)$ and the extracted best path is $\mathcal{P}_i$, all nodes lying on this path, having the same best source-sink pair

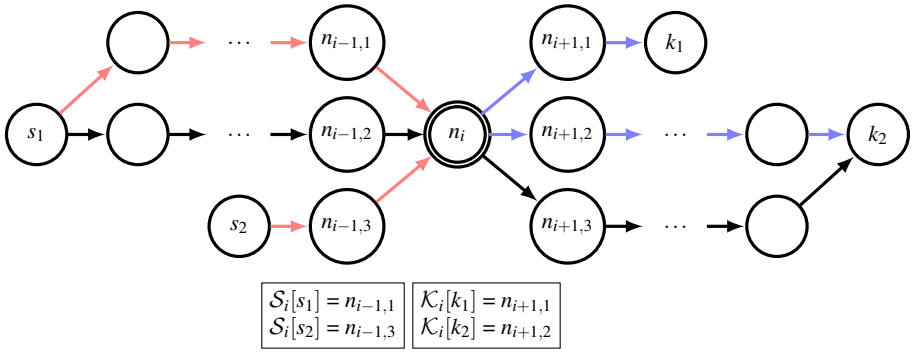


Fig. 4 Illustration of Algorithm 1. For every node $\mathbf{n}_i$ in the DAG, the lowest cost (and the corresponding best neighbor) to every reachable source is computed iteratively and stored in the associative map $\mathcal{S}_i$. In this figure, for example, the best path from n_i to source s_1 is via its neighbor $n_{i-1,1}$. Similarly lowest costs to all reachable sinks are stored in the map $\mathcal{K}_i$. After these values are computed, the best source-sink pair (s, k) is computed with the lowest cost using Eq. (8) and the best path $\mathcal{P}_{min}$ from s to k passing through n_i is traced out. All nodes lying on $\mathcal{P}_{min}$ which have the same best source-sink pair (s, k) need not be processed as the best path through any such node is $\mathcal{P}_{min}$ itself. The final output is the set of all paths passing through every node in the DAG.

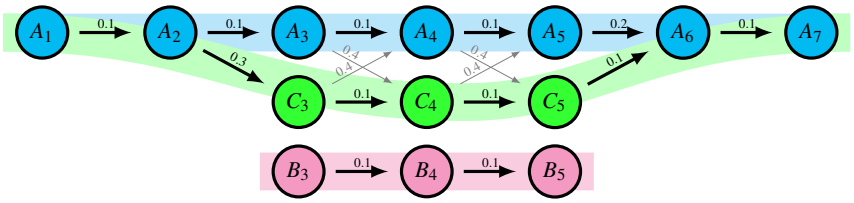


Fig. 5 The shortest paths through all nodes in the DAG combined represent our track graph structure. Our algorithm avoids computing the three shortest paths (given by the blue, green and magenta bands) for every single node naively, but instead traces a single shortest path only once. Nodes which lie on a shortest path and has the same source-sink pair, trivially trace the same path.

$(\mathbf{s}_i, \mathbf{k}_i)$, will trace out the exact same path. Hence, while determining unique paths in our solution, we can avoid tracing paths from all such nodes. See Figure 4 for an illustration.

After all nodes have been examined, we are left with the set of best paths passing through every single node in the DAG. An illustration of the output is shown in Figure 5.

Data: Set of all nodes $\mathbf{N} = \{\mathbf{n}_1, \mathbf{n}_2, \ldots, \mathbf{n}_m\}$ sorted topologically. Incoming nodes to n_i given by $\mathbf{I}_i$. The weight of an edge between nodes n_i and n_j is given by $d_{i,j}$.
Result: Set of associations of all reachable sources and the best path to them from every node n_i given by $\mathcal{S}_i$.

```
1  begin
2      for ∀n_i ∈ N do
3          for ∀n_j ∈ N do
4              if |I_j| = 0 then
5                  if n_i = n_j then
6                      S_i[j] ← n_j
7                      S_{i,cost}[j] ← 0
8                  else
9                      S_i[j] ← −1
10                     S_{i,cost}[j] ← ∞
11     for ∀n_i ∈ N do
12         for ∀n_j ∈ I_i do
13             for ∀source ∈ S_j do
14                 cost_new ← S_{j,cost}[source] ⊕ d_{i,j}
15                 if cost_new < S_{i,cost}[source] then
16                     S_i[source] ← n_j
17                     S_{i,cost}[source] ← cost_new
```

Algorithm 1: Algorithm to find the associations for the best routes to any node from all reachable sources. The best routes to all reachable sinks are determined by running the same algorithm with the nodes sorted in reverse order.

3.4 Complexity Analysis

Let us assume, without loss of generality, that the average number of features in every timestep is n. For t timesteps, we would then have a total of tn nodes in the entire DAG. The number of edges is bounded by n^2 between every pair of successive timesteps, so the total number of edges would be bounded by tn^2. The naïve version of the algorithm is a combination of two simple Dijkstra runs from any given node to all reachable sources and sinks. As we know the runtime of this algorithm is $O(V+E)$, for V vertices and E edges in a graph, the runtime in the naïve case will be $O(tn+tn^2)$ or $O(tn^2)$ for every node. Hence, if we were to run the naïve algorithm for all nodes, the runtime would be given by $O(t^2n^3)$ in the worst case.

Now for our improved algorithm, assuming the number of sources/sinks is given by p, we can safely say that $p << tn$. The runtime of Algorithm 1 is then given by $O(tnp+tn^2p)$ or $O(tn^2p)$. For Algorithm 2, it is $O(tnp^2+t^2n)$. So the total runtime of our algorithm would be given by $O(tn^2p+tnp^2+t^2n)$ which in practice (as seen in Table 1) is far less than $O(t^2n^3)$.

The memory footprint for the naïve version is bounded by the normal Dijkstra runtime of $O(N)$ for N nodes. Thus, in our scenario, it is given by $O(tn)$ as the shortest path via every node is computed independently. For the improved algorithm however, we need to store the mappings of shortest paths from all incoming/outgoing

Data: The sets of associated maps $\mathcal{S}_i$ and $\mathcal{K}_i$ for every node $\mathbf{n}_i$ as obtained from Algorithm 1. The function $tracePath(n_i, s, k)$ traces the path to source s and sink k from node n_i using the information present in $\mathcal{S}_i$ and $\mathcal{K}_i$.

Result: Set of all unique shortest paths $\mathbf{P}$ where there exists at least one path passing through every node n_i.

```
1  begin
2      P ← ∅
3      for ∀n_i ∈ N do
4          bestScore ← ∞
5          bestSource_i ← −1
6          bestSink_i ← −1
7          done_i ← false
8      for ∀n_i ∈ N do
9          for ∀s ∈ S_i do
10             for ∀k ∈ K_i do
11                 score ← s_cost ⊕ k_cost
12                 if score < best then
13                     bestScore ← score
14                     bestSource_i ← s
15                     bestSink_i ← k
16     for ∀n_i ∈ N do
17         if done_i ≠ true then
18             P ← tracePath(n_i, bestSource_i, bestSink_i)
19             done_i ← true
20             P ← P ∪ P
21             for ∀n_j ∈ P do
22                 if bestSource_j = bestSource_i ∧ bestSink_j = bestSink_i then
23                     done_j ← true
```

Algorithm 2: Algorithm to find shortest paths via every node

edges to all reachable sources/sinks and hence the memory footprint is given by $O(tnp)$.

3.5 Filtering similar paths for visualization

For visualization purposes we need to choose the best candidate paths which best represent a feature track at some spatio-temporal region.

In most cases, due to slight perturbations in the DAG, two unique paths may differ only at very few node positions with most of their nodes being identical. An example of this can be observed in Figure 5, where the blue and green paths show in essence the same structure with only a slight perturbation.

We aim to show the path with the best objective score, while other similar paths falling within a specified threshold are filtered out. The similarity g between two paths is estimated using

Dataset	Benzene	2D Checker-board	2D Streak Line Curvature	3D Square Cylinder
Dimensions	$127 \times 127 \times 255$	$128 \times 128 \times 1$	$750 \times 136 \times 1$	$192 \times 64 \times 48$
Time Steps	101	128	429	134
DAG Nodes	2239	3413	18035	15818
DAG Edges	2121	3198	85262	76739
Time to extract	10 ms	16 ms	1767 ms	3200 ms
Time to filter	5 ms	10 ms	80 ms	96 ms
Time naïve alg.	507 ms	936 ms	442050 ms	283448 ms
Memory	427 KB	758 KB	26322 KB	70814 KB
Memory naïve alg.	23 KB	12 KB	523 KB	372 KB

Table 1 Computation runtimes and memory requirements of our algorithm versus the naïve one for several data sets. All our experiments were performed on a machine with a 2.3GHz Intel i7 processor and 16GB main memory. Timings are totaled over the entire dataset. The timings do not include computation and simplification of the merge trees. It is to be noted that runtimes and memory usage depend *only* on the size of the DAG and not on the size of the dataset.

$$g(\mathcal{P}_1, \mathcal{P}_2) = \frac{|\mathcal{P}_1 \cap \mathcal{P}_2|}{\max(|\mathcal{P}_1|, |\mathcal{P}_2|)} \tag{9}$$

where $|\mathcal{P}_1 \cap \mathcal{P}_2|$ represent the number of matching edges. The function g estimates the fraction of edges that are identical in both paths. The filtration using function g is applied as follows. All paths obtained by solving Equation 8 for every node are sorted according to the best objective function score given by Equation 3. Paths are then processed in this sorted order, from lowest score to highest. If a path falls above the similarity threshold with any other path encountered before, it is filtered out. All other paths are retained.

If the filter node is set to be 100%, we are left with the complete set of unique paths. In our experiments, a filter rate of 70% shows best results.

4 Results

The timing and memory consumption for our method are given in Table 1. Regarding the computation times, note how our algorithm improves over the naïve version by up to two orders of magnitude. Regarding the memory consumption, the naïve method has lower memory usage as it only processes one node at once, while our algorithm processes all nodes together. Hence, considering the number of nodes in each data set, our algorithm is quite efficient with regards to memory usage as well.

Figure 6 shows a rotating and translating benzene data set. Since the data is not truly time-dependent, but just transformed rigidly, this serves as a test case to show that we capture all expected tracks and that our method is invariant against rotations and translations.

Figure 7 shows the 2D time-dependent Streak Line Curvature dataset.

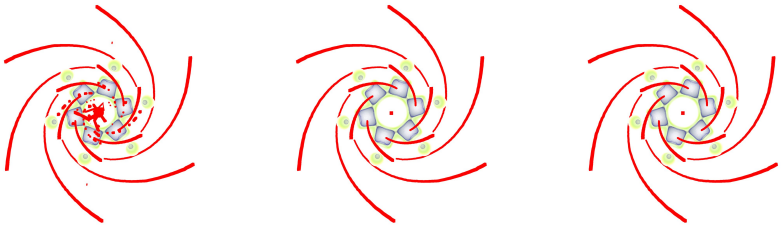


Fig. 6 Our method applied to the Benzene dataset. The paths indicate tracking of centers of mass of the regions signified by nodes in our DAG. (a) Paths of all lengths at filter rate 100% (b) Paths of length 100 and above at filter rate 100% (c) Paths of length 100 and above at filter rate 70%.

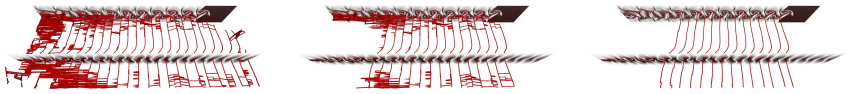


Fig. 7 (a) The 2D Streak Line Curvature dataset at filter rate 100% and showing paths of all lengths. (b) At 100% filter rate and full length paths only (c) At 70% filter rate and full length paths only.

Figure 8 shows the tracks for the smallest super/sub level set regions in a 2D Checkerboard dataset. The checkerboard pattern starts off smoothly and becomes increasingly noisy with time.

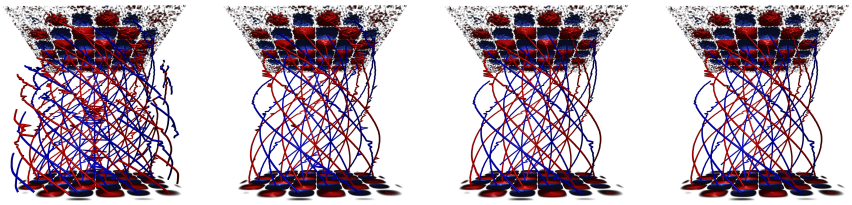


Fig. 8 Rotating 2D Checkerboard dataset. Tracks for the centers of mass of the smallest super/sub level sets are shown. (a) Filter rate 100% and paths of all lengths (b) Filter rate 100% and long (100 length or more) paths only (c) Filter rate 70% and long paths only (d) Filter rate 70% with long paths obtained from the naive version of the algorithm.

Figure 9 shows all the tracks in a flow around a 3D Square Cylinder. The location of the center of mass of a region is used to visualize the paths in all result images.

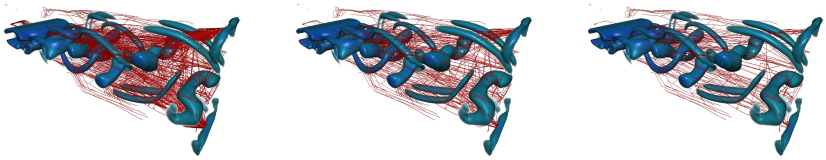


Fig. 9 Flow around a Square Cylinder dataset (a) Paths of all lengths extracted at filter rate of 100% (b) Paths of all lengths at filter rate 90% (c) Paths of all lengths at filter rate 70%.

5 Conclusion

We presented an extension of the method in [14] which was used to extract the best track through a chosen region at any given timestep in a time-dependent scalar field. These regions are based on topological segmentations in the spatial domain using merge trees and form the nodes in a Directed Acyclic Graph (DAG) structure in the spatio-temporal domain. Using the method in [14] to extract the best tracks through all nodes naïvely results in tracing the same paths multiple times. The algorithm presented in this paper makes use of the structure in the DAG to iteratively compute the best paths to every node from all reachable sources and sinks. This in turn allows us to compute the best paths through all nodes at orders of magnitude faster than the naïve approach. We also presented a filtering algorithm to filter out very similar paths for visualizing all paths together. Further work may include clustering these paths according to their similarity by using temporal similarity estimation techniques like dynamic time warping.

References

1. Bauer, D., Peikert, R.: Vortex tracking in scale-space. In: Proceedings of the Symposium on Data Visualisation 2002, VISSYM '02, pp. 233–ff. Eurographics Association, Aire-la-Ville, Switzerland, Switzerland (2002)
2. Bremer, P.T., Weber, G., Tierny, J., Pascucci, V., Day, M., Bell, J.: Interactive exploration and analysis of large-scale simulations using topology-based data segmentation. IEEE TVCG **17**(9), 1307–1324 (2011)
3. Bujack, R., Hotz, I., Scheuermann, G., Hitzer, E.: Moment invariants for 2d flow fields using normalization. In: 2014 IEEE Pacific Visualization Symposium, pp. 41–48 (2014)
4. Garth, C., Tricoche, X., Scheuermann, G.: Tracking of vector field singularities in unstructured 3d time-dependent datasets. In: Proceedings of the Conference on Visualization '04, VIS '04, pp. 329–336. IEEE Computer Society, Washington, DC, USA (2004)
5. Günther, T., Schulze, M., Theisel, H.: Rotation invariant vortices for flow visualization. IEEE TVCG **22**(1), 817–826 (2016)
6. Günther, T., Theisel, H.: Vortex cores of inertial particles. IEEE TVCG **20**(12), 2535–2544 (2014)
7. Kettner, L., Rossignac, J., Snoeyink, J.: The safari interface for visualizing time-dependent volume data using iso-surfaces and contour spectra. Computational Geometry **25**(1), 97 – 116 (2003). European Workshop on Computational Geometry - CG01

8. Oesterling, P., Heine, C., Weber, G.H., Morozov, D., Scheuermann, G.: Computing and visualizing time-varying merge trees for high-dimensional data. In: H. Carr, C. Garth, T. Weinkauf (eds.) Topological Methods in Data Analysis and Visualization IV, pp. 87–101. Springer International Publishing, Cham (2017)
9. Pele, O., Werman, M.: The quadratic-chi histogram distance family. In: K. Daniilidis, P. Maragos, N. Paragios (eds.) Computer Vision – ECCV 2010, pp. 749–762. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)
10. Reinders, F., Sadarjoen, I.A., Vrolijk, B., Post, F.H.: Vortex tracking and visualisation in a flow past a tapered cylinder. Computer Graphics Forum **21**(4), 675–682 (2002)
11. Reininghaus, J., Kasten, J., Weinkauf, T., Hotz, I.: Efficient computation of Combinatorial Feature Flow Fields. IEEE TVCG **18**(9), 1563–1573 (2012)
12. Saikia, H., Seidel, H.P., Weinkauf, T.: Extended branch decomposition graphs: Structural comparison of scalar data. Computer Graphics Forum (Proc. EuroVis) **33**(3), 41–50 (2014)
13. Saikia, H., Seidel, H.P., Weinkauf, T.: Fast similarity search in scalar fields using merging histograms. In: H. Carr, C. Garth, T. Weinkauf (eds.) TopoInVis, pp. 1–14. Annweiler, Germany (2015)
14. Saikia, H., Weinkauf, T.: Global feature tracking and similarity estimation in time-dependent scalar fields. Computer Graphics Forum **36**(3), 1–11 (2017)
15. Samtaney, R., Silver, D., Zabusky, N., Cao, J.: Visualizing features and tracking their evolution. Computer **27**(7), 20–27 (1994)
16. Sohn, B.S., Bajaj, C.: Time-varying contour topology. IEEE TVCG **12**(1), 14–25 (2006)
17. Szymczak, A.: Subdomain aware contour trees and contour evolution in time-dependent scalar fields. In: International Conference on Shape Modeling and Applications 2005 (SMI' 05), pp. 136–144 (2005)
18. Theisel, H., Sahner, J., Weinkauf, T., Hege, H.., Seidel, H..: Extraction of parallel vector surfaces in 3d time-dependent fields and application to vortex core line tracking. In: VIS 05. IEEE Visualization, 2005., pp. 631–638 (2005)
19. Theisel, H., Seidel, H.P.: Feature flow fields. In: Proceedings of the Symposium on Data Visualisation 2003, VISSYM '03, pp. 141–148. Eurographics Association, Aire-la-Ville, Switzerland, Switzerland (2003)
20. Thomas, D., Natarajan, V.: Multiscale symmetry detection in scalar fields by clustering contours. IEEE TVCG **20**(12), 2427–2436 (2014)
21. Thomas, D.M., Natarajan, V.: Symmetry in scalar field topology. IEEE TVCG **17**(12), 2035–2044 (2011)
22. Thomas, D.M., Natarajan, V.: Detecting symmetry in scalar fields using augmented extremum graphs. IEEE TVCG **19**(12), 2663–2672 (2013)
23. Tricoche, X., Wischgoll, T., Scheuermann, G., Hagen, H.: Topology tracking for the visualization of time-dependent two-dimensional flows. Computers & Graphics **26**(2), 249–257 (2002)
24. Wang, Z., Seidel, H.P., Weinkauf, T.: Multi-field pattern matching based on sparse feature sampling. IEEE TVCG (Proc. IEEE VIS) **22**(1), 807–816 (2016)
25. Weinkauf, T., Hege, H.C., Theisel, H.: Advected tangent curves: A general scheme for characteristic curves of flow fields. Computer Graphics Forum (Proc. Eurographics) **31**(2), 825–834 (2012). Eurographics 2012, Cagliari, Italy, May 13 - 18
26. Weinkauf, T., Sahner, J., Theisel, H., Hege, H.C.: Cores of swirling particle motion in unsteady flows. IEEE TVCG (Proc. IEEE Visualization) **13**(6), 1759–1766 (2007)
27. Weinkauf, T., Theisel, H., Gelder, A.V., Pang, A.: Stable Feature Flow Fields. IEEE TVCG **17**(6), 770–780 (2011)