On Computing Local Separators for Skeletonization[*]

J. Andreas Bærentzen, Rasmus E. Christensen, Emil Toftegaard Gæde, and Eva Rotenberg

Technical University of Denmark, Lyngby, Denmark
`{janba,etoga,erot}@dtu.dk`

Abstract

The notion of local separators for computing curve skeletons stems from the recent algorithm by Bærentzen and Rotenberg in [ACM Tran. Graphics'21]. Here the computation of such local separators plays an intrinsic role, with expensive computation becoming prohibitive for practical application to larger inputs.

In this work, we dive into these computations, examining and analysing in greater detail the individual steps, to clarify what bottlenecks exist for theoretical and empirical running times.

We give a simple modification to a phase of the computation, asymptotically improving the running time, and present empirical results that demonstrate the increase in practical performance.

1 Introduction

Curve skeletons are simplified, stick-like representations of shapes, that can be used for a wide variety of applications [7, 17, 4, 14, 20], and can be computed through a wide variety of approaches [8, 16, 3, 19, 13, 22, 15, 21, 1, 6, 11].

J.A. Bærentzen and E. Rotenberg propose in [2] a new algorithm for computing a curve skeleton using local separators, which we will refer to as the *Local Separator Skeletonization* algorithm, or simply LSS. The LSS algorithm seemingly generates output of high quality, while making relatively little assumptions about the input, requiring it only to be a spatially embedded graph. It works by a three phased approach, as seen in Figure 1, in which first a number of minimal local separators are computed, then a set of non-overlapping separators are selected, and finally a skeleton is extracted.

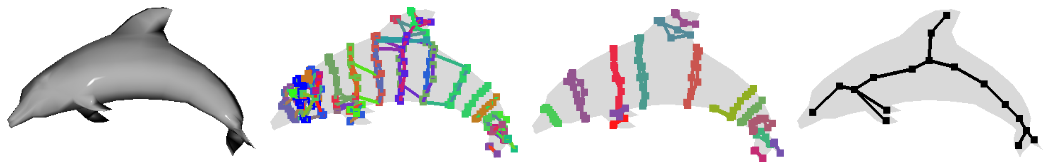


Figure 1 Visualisation of the three phases of the LSS algorithm. From left to right: A shaded render of the input, a number of computed minimal separators, a non-overlapping subset of the separators, and the resulting skeleton after extraction.

The LSS algorithm has the drawback that finding minimal local separators is computationally costly, requiring the input to be simplified in order for the running time not to be prohibitive. In this paper we give a brief analysis of the cost of computing local separators,

[*] *Emil Toftegaard Gæde* and *Eva Rotenberg* are supported by Eva Rotenberg's Carlsberg Foundation Young Researcher Fellowship CF21-0302 - "Graph Algorithms with Geometric Applications"

39th European Workshop on Computational Geometry, Barcelona, Spain, March 29–31, 2023.
This is an extended abstract of a presentation given at EuroCG'23. It has been made public for the benefit of the community and should be considered a preprint rather than a formally reviewed paper. Thus, this work is expected to appear eventually in more final form at a conference with formal proceedings and/or in a journal.

examine practical bottlenecks, and propose a simple modification that we show improves the running time of LSS, without altering the output.

1.1 Preliminaries

We consider a spatially embedded straight line graph, $G = \langle V, E \rangle$, with no assumptions about the origin of the graph, such as whether it is sampled from the surface of a manifold, created from a point cloud, or otherwise. If G is not connected, we simply run our algorithms on each component separately, thus we assume G to be connected.

In graph theory, an induced subgraph $G[V']$, is the graph $G' = \langle V', E' \rangle$ where E' is a subset of E that contains any edge where both endpoints are in V'. We define the closed neighbourhood of a vertex, $v \in V$, denoted $N(v)$, to be set of vertices adjacent to v and v itself. For a set of vertices, $S \subseteq V$, we define the neighbourhood as $N(S) = \bigcup N(s), s \in S$.

A *vertex separator* is a subset of vertices whose removal disconnects the graph. A minimal vertex separator is a separator where no proper subset is itself a separator. In [2], this notion is extended to *local separators*, defined as a subset of vertices, $S \subset V$, that is a vertex separator of $G[N(S)]$. Intuitively, we cannot remove a vertex from a *minimal local separator* without the remaining set ceasing to be a local separator. Formally we define a minimal local separator as $S \subset V$ s.t. S is a separator of $G[N(S)]$ and there exists no subset $S' \subset S$ s.t. S' is a separator of $G[N(S')]$. We note that S' does not need to separate the same induced subgraph as S.

2 Computing Local Separators

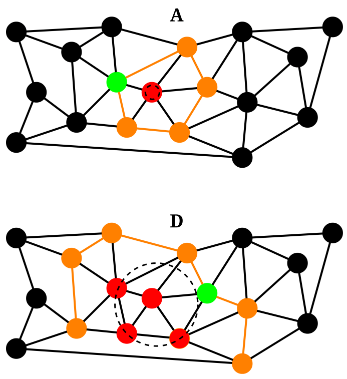
We recall the local separator construction algorithm of LSS, and supply a theoretical analysis of the worst case running time. With this in mind we then perform an empirical examination, detailing how the running time is distributed amongst the phases of computation in practice. Finally, we use this analysis to show our improvement to the running time of the algorithm, both asymptotically and empirically.

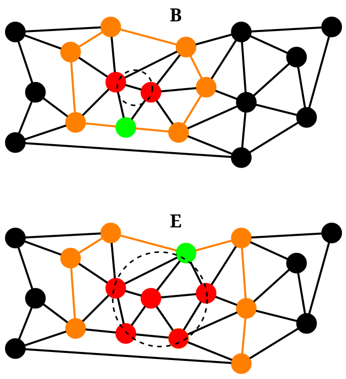
The algorithm for computing local separators is a heuristic algorithm that, intuitively, captures structural features of the input. It works in two phases. First, given a vertex $v \in V$, a local separator is constructed as follows (Figure 2): we maintain a candidate separator Σ, and what is called the front $F = N(\Sigma) \setminus \Sigma$. Additionally, we maintain some bounding sphere $B(\Sigma)$ that contains Σ (and possibly other vertices in V).

Initially, Σ contains v, F contains the vertices adjacent to v, and the bounding sphere has its centre at v and radius 0. We then iteratively pick, from F, the vertex closest to the centre of the bounding sphere, say s, and add it to Σ. We also remove it from F and add all neighbours of s not in Σ to F, and then update the bounding sphere to encapsulate s.

This procedure is repeated until the graph induced by F consists of more than one connected component (i.e. Σ is a local separator of $G[N(\Sigma)]$). The process is visualised in Figure 2, with (A) showing the initial configuration, (B-E) showing the iterative growing of the separator, and (F) showing the final separator after disconnecting the front.

We subsequently elaborate on shrinking a separator that has been constructed in the previous phase. First the vertices of Σ are ordered from high to low according to the distance from the centre of $B(\Sigma)$ to each vertex, after performing a Laplacian smoothing of Σ. We then perform a pass over Σ and move any vertices that are adjacent to exactly one component of F, but whose removal from Σ would not reconnect F, to that adjacent component. After such a linear scan over Σ, we restart the process (as removing vertices later in the order,





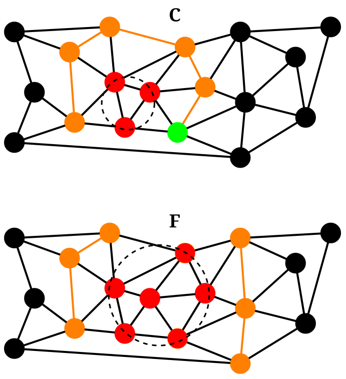


Figure 2 The process of growing a separator. Red vertices are those currently in the separator, orange vertices and edges are those currently in the front, and green vertices are the vertices of the front that are closest to the centre of the bounding sphere, which is indicated by the dotted circle. This figure is heavily inspired by Figure 6 of [2].

may make vertices earlier in the order eligible for removal), until no vertices are able to be removed (i.e. Σ is a minimal local separator).

We finally analyse the complexity of computing a local separator. Let V_F, E_F denote the vertices and edges of the F respectively, and consider each iteration of the growing phase:

1. Find closest $v \in F$ to $B(\Sigma)$	$O(\lvert V_F\rvert)$ by linear scan through V_F
2. Move v to Σ and update $B(\Sigma)$	$O(1)$
3. Add $N(v) \setminus \Sigma$ to V_F	$O(\textsc{deg}(v))$
4. Check if F is connected	$O(\lvert E_F\rvert)$ by breadth first search through F

Let Σ^* denote the result of the growing phase. We spend $|\Sigma^*|$ iterations growing, and in the worst case Σ^* and F are proportional to G s.t. $|\Sigma^*| = O(|V|), |V_F| = O(|V|), |E_F| = O(|E|)$. Thus we spend a total of $O(|V|^2)$ time on step 1, $O(|V|)$ time on step 2, $O(|E|)$ time on step 3, and $O(|V||E|)$ on step 4. The total running time spent in the growing phase, in the worst case, is then $O(|V|^2 + |E||V|)$.

To shrink the separator we compute a smoothing of positions, by considering for each vertex the positions of its neighbours, and order the vertices accordingly. Having smoothed and sorted Σ^*, the procedure then iterates over Σ^* and removes vertices until Σ^* is minimal:

5. Compute smoothing of Σ^*	$O(\textsc{deg}(s))$ for all $s \in \Sigma^*$ totalling $O(E)$
6. Order Σ^* according to smoothed positions	$O(\lvert\Sigma^*\rvert \log \lvert\Sigma^*\rvert)$
7. Iterate over $s \in \Sigma^*$ in their sorted order	$O(\lvert\Sigma^*\rvert)$ total time
▪ Check if s is incident to exactly one connected component of F.	$O(1)$
▪ If it is, remove it from Σ^* and inform $N(v) \cap \Sigma^*$	$O(\textsc{deg}(v))$, uniquely charged to v.
8. Repeat step 7 until Σ^* is minimal	$O(\lvert\Sigma^*\rvert)$ repetitions

In the worst case we remove only a constant number of vertices in each iteration of step 7, requiring $O(|\Sigma^*|)$ passes over $O(|\Sigma^*|)$ vertices, totalling $O(|\Sigma^*|^2)$ time. When removing a vertex v from Σ^* we inform the neighbours in Σ^* that they are now adjacent to the connected component of F that v was moved to. This incurs at most a cost of $O(\textsc{deg}(s))$ for all $s \in \Sigma^*$ totalling $O(E)$. The worst case running time for shrinking is then $O(|V|^2 + |E|)$, and the total time to compute a minimal local separator, including both growing and shrinking, in the worst case, is then $O(|V|^2 + |V||E|)$.

In order to further our understanding of what makes the computation slow in practice, we perform a series of measurements that also include details about how time is spent in each step.

Input	Vertices	Edges	Step 1 (s)	Step 3 & 4 (s)	Step 5 & 6 (s)	Step 7 & 8 (s)
`wsm0.25`	4946	14856	0.29	19.1	2.11	0.33
`wsm0.5`	9898	29712	1.38	89.9	10.3	1.17
`wsm1`	19803	59427	9.19	589.4	89.9	5.7
`wsv60`	6166	63136	1.50	193.6	17.8	1.2
`wsv90`	20966	233615	34.9	4372.9	478.9	14.4

Table 1 Measurements of the phases of computing a separator on a small sample of increasingly simplified meshes (`wsm*`) and voxel grids (`wsv*`) all constructed from the `wooden_statue`.

From our resulting measurements, shown in Table 1, we see that the dominating is step 4 (checking for connectivity), making it the most fitting candidate for optimisation. In Figure 3 we show the `wooden_statue` used to generate `wsm*` and `wsv*` respectively, as well as computed non-overlapping separators on both mesh and voxel grid.

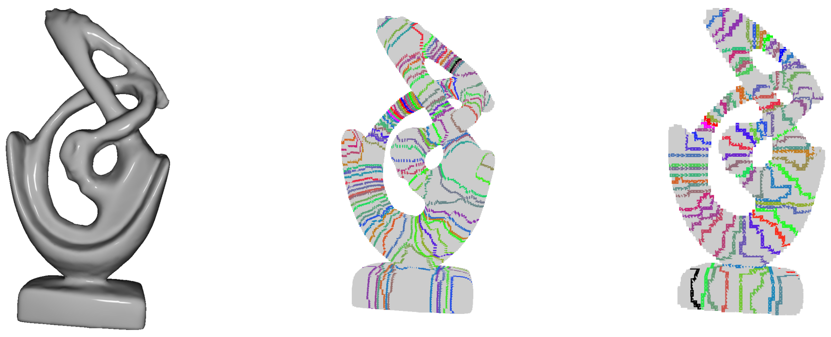


Figure 3 Left: the `wooden_statue` input used to generate `wsm*` and `wsv*`. Centre: separators computed on `wsm1`. Right: separators computed on `wsv90`.

3 Dynamic Connectivity

The simplest and most intuitive solution to the fully dynamic connectivity problem is the use of augmented Euler Tour Trees [9]. These augmented trees give $O(\log |V|)$ insertions, $O(|V| \log |V|)$ deletions and $O(\log |V|)$ connectivity queries.

In order to improve and bring into balance the update times, one can use the dynamic connectivity data structure of Holm, de Lichtenberg and Thorup [10]. It is directly cited in [2] as a suggestion, and it has been previously examined in practice [12]. We refer to this data structure as the HLT data structure.

The HLT data structure maintains a hierarchical decomposition of forests of augmented Euler Tour Trees, using the levels of the hierarchy to limit redundant searches when reconnecting. This allows for updates in amortized $O(\log^2 |V|)$ time [10].

To integrate the HLT data structure into our computations of local separators, we simply maintain the front as part of the iterative growing. The complexity of growing is then:

1. Find closest $v \in F$ to $B(\Sigma)$ $O(|V_F|)$ by linear scan through V_F

2. Move v to Σ and update $B(\Sigma)$ $O(1)$
3. Add $N(v) \setminus \Sigma$ to F $O(\mathrm{DEG}(v) \log^2 |V_F|)$ amortized using HLT.
4. Check if F is connected $O(1)$ by maintaining the number of components in HLT

The total time becomes $O(|V|^2 + |E| \log^2 |V|)$ by the same observations as previously.

In practice, it is suggested [12] to limit the height of the hierarchy of the dynamic connectivity data structure, by not using the hierarchy, once forests reach small size. We have implemented and integrated the structure into our computations, and measured that the optimal threshold is one that is large enough to effectively eliminate the use of the hierarchy.

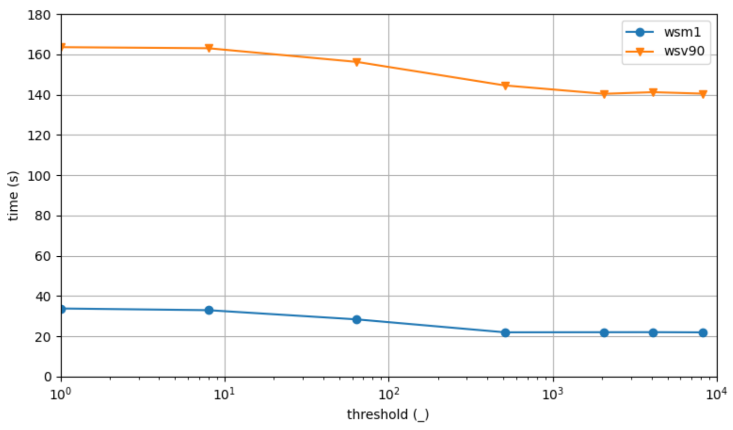


Figure 4 Time spent computing separators for a mesh (`wsm1`) and a voxel grid (`wsv90`) as a function of threshold value for the dynamic connectivity data structure.

As shown in Figure 4, the running time decreases as the threshold increases up to a certain point before flattening out. At this point, the threshold is large enough that the hierarchy consists of a single level, essentially reducing the structure to simply an augmented forest of Euler Tour Trees.

4 Empirical Results

Here we present our main result, showing how our variation compares to LSS in terms of running time. All values shown are the medians of three runs on a HP Elitebook 840 G8 with an i7 processor. The source code is publicly available through the GEL repository [5] and is compiled using `O3` optimisations. In Figure 5, we compare the old method to the one using dynamic connectivity (dyn), and find that for both meshes and voxels there is a large improvement. It is worth noting that this improvement seems even greater for voxel grids.

However, we also examine running times on a more refined scale, in order to examine how the relation between phases has changed, as seen in Table 2.

Of note is that not only has the check for connectivity been drastically reduced, especially for voxel grids, but it also seems that the bottleneck has shifted in this case towards shrinking.

In addition to measuring on `wooden_statue`, we have also measured the time spent checking connectivity on the Groningen Skeletonization Benchmark [18], the results of which can be seen in Figure 6. We note that input for which the old method exceeded half an hour was left out.

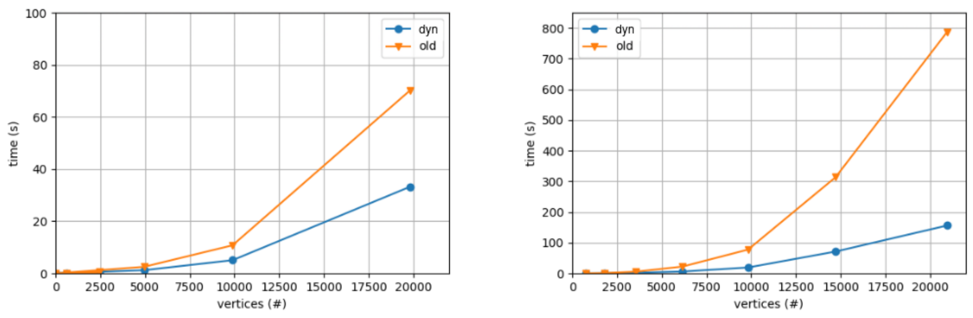


Figure 5 Running times of local separator computations without (old) and with (dyn) the use of dynamic connectivity. Left shows performance on meshes while right shows performance on voxels.

Input	Vertices	Edges	Step 1 (s)	Step 3 & 4 (s)	Step 5 & 6 (s)	Step 7 & 8 (s)
`wsm0.25`	4946	14856	0.21	4.28	1.43	0.22
`wsm0.5`	9898	29712	1.05	17.2	7.78	0.85
`wsm1`	19803	59427	8.68	109.1	80.8	5.08
`wsv60`	6166	63136	1.4	22.7	16.2	1.07
`wsv90`	20966	233615	21.9	214.7	313.9	9.75

Table 2 Measurements of the phases of computing a separator on a small sample of meshes (wsm*) and voxel grids (wsv*) all of similar structure, using the dynamic connectivity data structure.

Acknowledgements

The authors would like to thank Ivor van der Hoog for his extensive and useful feedback.

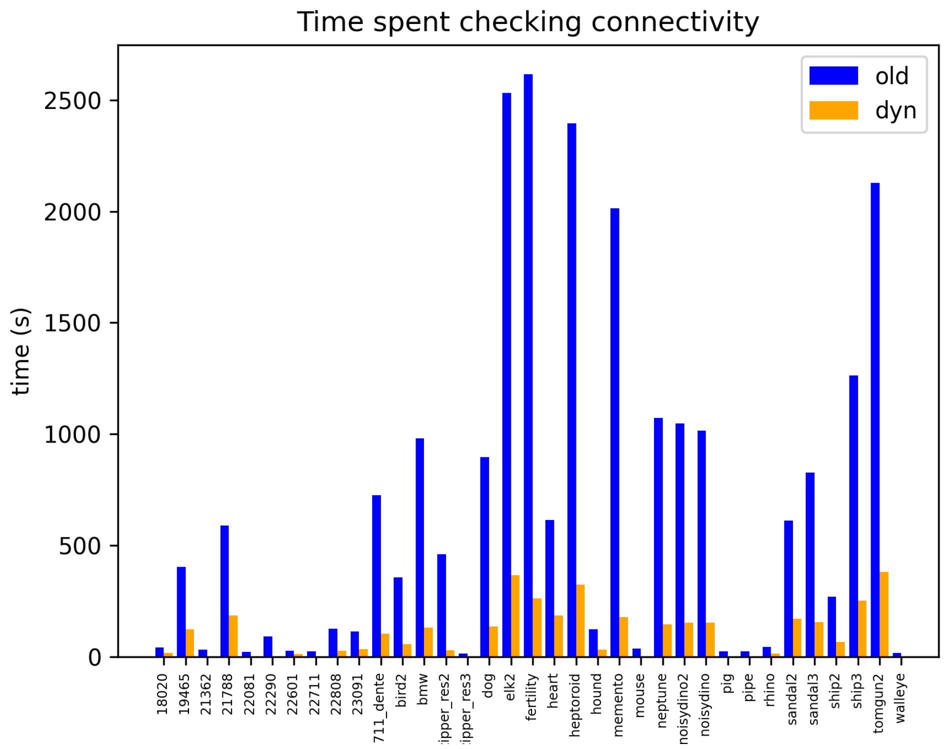


Figure 6 Time spent checking connectivity on input from the Groningen Skeletonization Benchmark.

References

1 Oscar Kin-Chung Au, Chiew-Lan Tai, Hung-Kuo Chu, Daniel Cohen-Or, and Tong-Yee Lee. Skeleton extraction by mesh contraction. *ACM Trans. Graph.*, 27(3):44, 2008. `doi:10.1145/1360612.1360643`.

2 Andreas Bærentzen and Eva Rotenberg. Skeletonization via local separators. *ACM Trans. Graph.*, 40(5):187:1–187:18, 2021. `doi:10.1145/3459233`.

3 Harry Blum. A transformation for extracting new descriptions of shape. *Models for the perception of speech and visual form*, pages 362–380, 1967.

4 Angela Brennecke and Tobias Isenberg. 3d shape matching using skeleton graphs. In Thomas Schulze, Stefan Schlechtweg, and Volkmar Hinz, editors, *Simulation und Visualisierung 2004 (SimVis 2004) 4-5 März 2004, Magdeburg*, pages 299–310. SCS Publishing House e.V., 2004. URL: `http://wwwisg.cs.uni-magdeburg.de/graphik/pub/files/Brennecke_2004_3SM.pdf`.

5 J. Andreas Bærentzen. Gel library, 2022. URL: `https://github.com/janba/GEL`.

6 Junjie Cao, Andrea Tagliasacchi, Matt Olson, Hao Zhang, and Zhixun Su. Point cloud skeletons via laplacian based contraction. In *SMI 2010, Shape Modeling International Conference, Aix en Provence, France, June 21-23 2010*, pages 187–197. IEEE Computer Society, 2010. `doi:10.1109/SMI.2010.25`.

7 Nicu D. Cornea, Deborah Silver, and Patrick Min. Curve-skeleton properties, applications, and algorithms. *IEEE Trans. Vis. Comput. Graph.*, 13(3):530–548, 2007. `doi:10.1109/TVCG.2007.1002`.

8 Tamal K. Dey and Jian Sun. Defining and computing curve-skeletons with medial geodesic function. In Alla Sheffer and Konrad Polthier, editors, *Proceedings of the Fourth Eurographics Symposium on Geometry Processing, Cagliari, Sardinia, Italy, June 26-28, 2006*, volume 256 of *ACM International Conference Proceeding Series*, pages 143–152. Eurographics Association, 2006. doi:10.2312/SGP/SGP06/143-152.

9 Monika Henzinger and Valerie King. Randomized dynamic graph algorithms with polylogarithmic time per operation. *Journal of the ACM*, 46:502–516, 01 1995. doi:10.1145/225058.225269.

10 Jacob Holm, Kristian de Lichtenberg, and Mikkel Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *J. ACM*, 48(4):723–760, 2001. doi:10.1145/502090.502095.

11 Hui Huang, Shihao Wu, Daniel Cohen-Or, Minglun Gong, Hao Zhang, Guiqing Li, and Baoquan Chen. L_1-medial skeleton of point cloud. *ACM Trans. Graph.*, 32(4):65:1–65:8, 2013. doi:10.1145/2461912.2461913.

12 Raj D. Iyer, David Karger, Hariharan S. Rahul, and Mikkel Thorup. An experimental study of poly-logarithmic fully-dynamic connectivity algorithms. *Acm Journal of Experimental Algorithmics*, 6:4, 2001. doi:10.1145/945394.945398.

13 Andrei C. Jalba, Jacek Kustra, and Alexandru C. Telea. Surface and curve skeletonization of large 3d models on the GPU. *IEEE Trans. Pattern Anal. Mach. Intell.*, 35(6):1495–1508, 2013. doi:10.1109/TPAMI.2012.212.

14 Diane Perchet, Catalin I. Fetita, and Françoise J. Prêteux. Advanced navigation tools for virtual bronchoscopy. In Edward R. Dougherty, Jaakko Astola, and Karen O. Egiazarian, editors, *Image Processing: Algorithms and Systems III, San Jose, California, USA, January 18, 2004*, volume 5298 of *SPIE Proceedings*, pages 147–158. SPIE, 2004. doi:10.1117/12.533096.

15 Dennie Reniers, Jarke J. van Wijk, and Alexandru C. Telea. Computing multiscale curve and surface skeletons of genus 0 shapes using a global importance measure. *IEEE Trans. Vis. Comput. Graph.*, 14(2):355–368, 2008. doi:10.1109/TVCG.2008.23.

16 Andrea Tagliasacchi, Ibraheem Alhashim, Matt Olson, and Hao Zhang. Mean curvature skeletons. *Comput. Graph. Forum*, 31(5):1735–1744, 2012. doi:10.1111/j.1467-8659.2012.03178.x.

17 Andrea Tagliasacchi, Thomas Delamé, Michela Spagnuolo, Nina Amenta, and Alexandru C. Telea. 3d skeletons: A state-of-the-art report. *Comput. Graph. Forum*, 35(2):573–597, 2016. doi:10.1111/cgf.12865.

18 Alexandru Telea. 3d skeletonization benchmark. https://webspace.science.uu.nl/~telea001/Shapes/SkelBenchmark, 2016. Accessed: 2022-10-31.

19 Alexandru C. Telea and Andrei C. Jalba. Computing curve skeletons from medial surfaces of 3d shapes. In Hamish A. Carr and Silvester Czanner, editors, *Theory and Practice of Computer Graphics, Rutherford, United Kingdom, 2012. Proceedings*, pages 99–106. Eurographics Association, 2012. doi:10.2312/LocalChapterEvents/TPCG/TPCG12/099-106.

20 Ming Wan, Frank Dachille, and Arie E. Kaufman. Distance-field-based skeletons for virtual navigation. In Thomas Ertl, Kenneth I. Joy, and Amitabh Varshney, editors, *12th IEEE Visualization Conference, IEEE Vis 2001, San Diego, CA, USA, October 24-26, 2001, Proceedings*, pages 239–246. IEEE Computer Society, 2001. doi:10.1109/VISUAL.2001.964517.

21 Yu-Shuen Wang and Tong-Yee Lee. Curve-skeleton extraction using iterative least squares optimization. *IEEE Trans. Vis. Comput. Graph.*, 14(4):926–936, 2008. doi:10.1109/TVCG.2008.38.

22 Yajie Yan, Kyle Sykes, Erin W. Chambers, David Letscher, and Tao Ju. Erosion thickness on medial axes of 3d shapes. *ACM Trans. Graph.*, 35(4):38:1–38:12, 2016. `doi:10.1145/2897824.2925938`.