Multi-Scale Graph-Based Skeletonization using Local Separators

Rasmus E. Christensen
Emil T. Gæde

Kongens Lyngby 2022

Technical University of Denmark
Department of Applied Mathematics and Computer Science
Richard Petersens Plads, building 324,
2800 Kongens Lyngby, Denmark
Phone +45 4525 3031
compute@compute.dtu.dk
www.compute.dtu.dk

Contents

1 Abstract **3**

2 Introduction **3**

3 Related Work **5**

4 Preliminaries **5**
4.1 Local Separators and the LSS Algorithm . . . 5
4.1.1 Growing Local Separators . . . 6
4.1.2 Minimizing Separators . . . 7
4.1.3 Separator Packing . . . 8
4.1.4 Skeleton Extraction . . . 9
4.2 Multi-Scale Graph . . . 9

5 Testing Method/Benchmarking **10**
5.1 Hardware and Language . . . 10
5.2 Test Data . . . 10
5.3 Testing . . . 10
5.3.1 The Baseline Algorithm . . . 11
5.3.2 Skeleton-quality . . . 11
5.4 Runtime testing . . . 12

6 Multi-Scale Approach for Skeletonization **12**
6.1 Generating Multi-Scale Graphs . . . 12
6.1.1 Analysis . . . 14
6.2 Finding Restricted Local Separators. . . . 15
6.2.1 Analysis . . . 16
6.3 Algorithm 1: Sample Starting Vertices (recomp) . . . 16
6.3.1 Description . . . 17
6.3.2 Analysis . . . 18
6.3.3 Implementation . . . 18
6.3.4 Discussion . . . 18
6.4 Algorithm 1 Variation: Growing From Static Centre (recomp-static) . . . 20
6.4.1 Description . . . 21
6.4.2 Analysis . . . 21
6.4.3 Implementation . . . 21
6.4.4 Discussion . . . 21
6.5 Algorithm 2: Separator Expansion (expand) . . . 22
6.5.1 Description . . . 23
6.5.2 Analysis . . . 23
6.5.3 Implementation . . . 24
6.5.4 Discussion . . . 24
6.6 Algorithm 2 Variation: Continuous Packing (expand-continuous) . . . 26
6.6.1 Description . . . 26
6.6.2 Analysis . . . 26
6.6.3 Discussion . . . 26
6.7 Algorithm 2 Variation: Capacity Packing (expand-capacity) . . . 27
6.7.1 Description . . . 28

6.7.2 Analysis 28
6.7.3 Discussion 28

7 Potential Improvements **30**
7.1 Multi-Scale Graph Level Count 30
7.2 Vertex Sampling 31

8 Results and Discussion **33**

9 Future Work **38**

10 Conclusion **39**

References **40**

A Appendix **41**

B Efficient Methods for Graph-Based Generation of Skeletons from 3D Data **45**

1 Abstract

A recent contribution to the field of skeletonization is an algorithm built on the notion of local separators. The local separator skeletonization algorithm proposed by J.A. Bærentzen and E. Rotenberg generates output of high quality, but is expensive to compute in practice.

In this project we examine how the algorithm can be adapted for a multi-scale approach, greatly improving computational performance while keeping a high quality output.

We describe and analyse the algorithm and its phases in detail, explore different ways of adapting it for multi-scale computation. We also analyse, implement and benchmark their performance.

The results show a huge increase in computational performance, but also caveats regarding the quality of the output in certain cases. We present our observations and suggestions for circumventing these issues, as well as other areas of interest for future work.

2 Introduction

Many different fields make use of 3D shapes, such as CAD, computer graphics, and medical imaging. Although the shape itself is often of interest, sometimes a compact and simplified representation is desirable. Examples of applications that use such representations include animation, shape recognition and more[1].

A mathematically well defined and intuitive notion is that of medial surfaces. Formally, the medial surface is the set of centre points of maximal inscribed balls in the shape. Informally, if we think of the shape as made from flammable material and we set fire to the entire boundary, the medial surface consists of the points where flames meet as the shape burns. Although well-defined and seemingly a

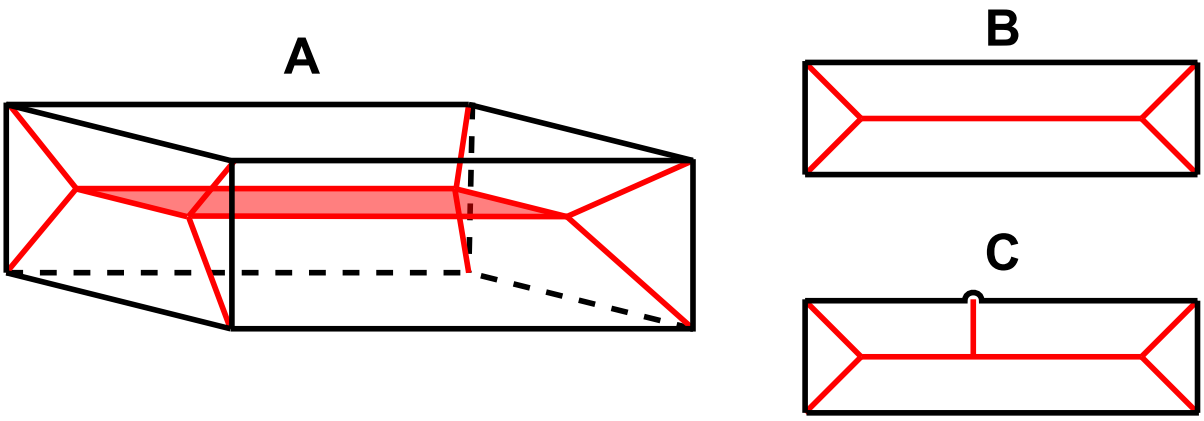


Figure 1: The medial surface of a rectangular slab (A,B) as well as the medial surface after introduction of a slight bump in the surface (C). This figure is heavily inspired by figure 1. of [1]

good candidate for a compact representation, the medial surface consists of surfaces, not curves, and is very sensitive to noise and small variations on the surface of the shape it represents. On figure 1 (A) the medial surface of a rectangular slab is shown, while (B) and (C) shows a side view and how a small bump may heavily influence the medial surface.

An alternative to medial surfaces that aims to work around these factors of medial surfaces is *curve skeletons*, as seen on figure 2.

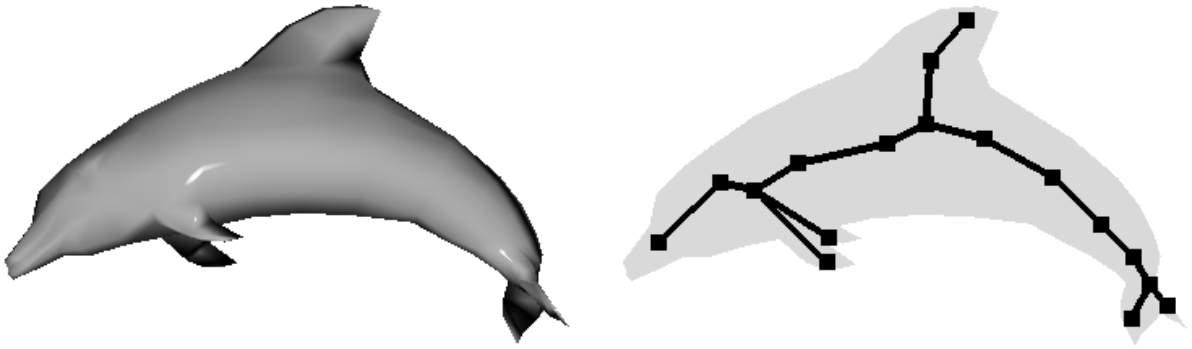


Figure 2: A shaded render of a dolphin, and a curve skeleton of the shape.

A curve skeleton is a representation consisting of curves that captures the features of the shape. An exact definition is not widely agreed upon, but Cornea et. al. gives the elusive definition "a simplified 1D representation of a 3D object" and presents desirable properties of curve skeletons[1]. Some of the properties include that the skeleton be:

- *Topology preserving.* The skeleton should preserve the topology of the shape, in the sense that topological holes and handles should be represented as loops in the skeleton.
- *Thin.* The skeleton should be 1D, consisting only of curves.
- *Centered.* The skeleton should lie close to the medial surface.
- *Robust.* The skeleton should be resistant to noise along the surface of the shape.
- *Reliable.* It should be possible to connect any point on the surface of the shape to the skeleton by a straight line that doesn't cross any boundaries.

In addition, the algorithm used to compute the skeleton should be efficient. Note that some of these properties are conflicting, ie. if the skeleton strictly adheres to the centering-property, it will have the same problems that medial surfaces face regarding noise, meaning it is not robust. How important a property is depends highly on the application, and various approaches to curve skeletons thus do not necessarily place emphasis on every property. For application in virtual navigation for instance, moving along a centered skeleton guarantees no collision with the shape[1], meaning the skeleton should prioritise centeredness over robustness.

A recent contribution to the field of skeletonization is the "Local Separator Skeletonization"-algorithm[2] (from hereon referred to as the LSS algorithm) that differentiates itself from previous methods by being based on the graph theory notion of separators. The LSS algorithm does not require input to be a specific surface representation, such as a triangle mesh, but rather a spatially embedded graph. This makes the method applicable to voxel-grids, meshes, graphs created from point clouds or even graphs that do not necessarily represent shapes. Another benefit of the LSS algorithm is that it seems to be able to capture small features missed by contractive measures while remaining robust.

However, the LSS algorithm is also computationally expensive, limiting its practical applications. In an earlier project (appendix B) we examined bottlenecks of the LSS algorithm, with the goal of improving performance without making significant algorithmic changes. Although we were able to improve the time spent on computing separators, we also found that algorithmic changes were necessary to make vast improvements.

In scientific computing, one approach to problems that are too computationally complex to solve con-

ventionally is to apply multi-scale algorithms[3]. Multi-scale algorithms work by solving subproblems at lower resolution, which saves computation, and then using these results to solve problems at higher resolution.

When reducing the resolution of shapes, small features are lost while large features become small. Thus, we should be able to capture features of all sizes by searching only for small features on various resolutions of the input.

The goal of this project is thus to examine how to apply a multi-scale approach to the LSS algorithm in order to efficiently skeletonize at a high level of quality.

3 Related Work

In the article "Skeletonization via Local Separators" J.A. Bærentzen and E. Rotenberg present the algorithm that this project builds upon[2]. The article gives an in-depth explanation of the algorithm, as well as reflection on the quality of the resulting skeletons and how they compare to other methods of skeletonization. It is worth noting that this project works on a slightly altered algorithm than the one presented in the article, which is explained in greater detail in section 4.1.

In appendix B we have included a previous project, "Efficient Methods for Graph-Based Generation of Skeletons from 3D Data". The project was a special course at DTU with the goal of examining how the implementation of the LSS algorithm available in the GEL library could be improved in terms of practical running time. The project includes detailed benchmarks of the phases of the LSS algorithm, as well as motivation and a technical explanation of the changes made to the LSS algorithm described in the original article.

In "Curve-Skeleton Properties, Application, and Algorithm" by Cornea et. al[1], we find not only the definition of curve skeletons with which we concern ourselves, but also reflections on their desired properties motivated by applications. The article provides additional examples, properties, descriptions of general approaches to the skeletonization problem, and a survey of how these approaches perform in terms of various properties.

In "Geometric Modeling Based on Polygonal Meshes" by M.Botsch et al.[4], various aspects of geometry processing based on polygonal meshes is described. Of specific interest is chapter 9 that covers mesh decimation and simplification. When applying a multi-scale approach we would like to reduce the complexity of the graphs while preserving their spatial structure. The chapter covers various approaches to simplification, as well as comments on how decimation should be performed to maintain topological correctness.

4 Preliminaries

4.1 Local Separators and the LSS Algorithm

The LSS algorithm builds on the concept of local separators. In graph theory, a separator is a set of vertices whose removal disconnects the graph into two (or more) non-empty connected components. This notion is then extended to *local* separators, whose removal disconnects the neighbourhood of the separator, rather than the entirety of the graph. In addition, we define minimal local separators to be local separators where if any vertex is removed, the set ceases to be a local separator. In the context of the remainder of this project, we use the term *separator* interchangeably with *local separator*.

We will first describe the overall approach and intuition behind the algorithm before describing in greater detail how each phase is performed.

The algorithm works in three phases, as shown on figure 3 (B-D). First we compute a set of (possibly overlapping) minimal local separators. We then perform packing to select a non-overlapping set of separators from which we extract the skeleton.

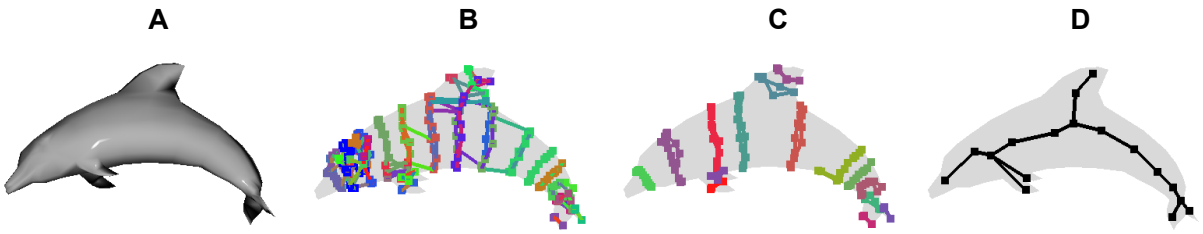


Figure 3: A visual representation of the phases of the LSS algorithm. The input (A), a computed set of minimal separators (B), a packed set of non-overlapping minimal separators (C), and the extracted skeleton (D).

To compute a minimal local separator we pick some starting vertex, grow a separator as described in section 4.1.1, and then shrink the separator, as described in section 4.1.2, to make it minimal.

To save computation, the algorithm uses a sampling scheme to select which vertices to grow separators from. A random ordering of the vertices is generated and then iterated over, and a separator is started from each vertex with probability 2^{-x} where x is the number of minimal separators currently computed that contain that vertex.

Packing is then performed on the found minimal separators, using a greedy set-packing approach described in section 4.1.3, and the skeleton is extracted as described in section 4.1.4. We note that this project makes no modifications to skeleton extraction, only the process used to find and pack the minimal separators.

In the following sections we will describe time complexities given a graph $G = \langle V, E \rangle$, a separator, Σ, and the subgraph induced by the neighbourhood of Σ, which we will denote $N(\Sigma) = \langle V', E' \rangle$. When stating complexities we use Σ, V, E, V' and E' to denote the cardinality of the respective sets.

We note that the general worst case runtime of computing and packing separators using LSS is $O(V^3 + VE \log^2 V)$, since we grow and shrink $O(V)$ separators in $O(V^2 + E \log^2 V)$ time before packing them in $O(V^3)$ time, as will be shown in the following sections.

4.1.1 Growing Local Separators

When computing a local separator, the LSS algorithm uses a region growing approach. Starting from some vertex we maintain two sets of vertices, Σ and F, respectively representing the separator and the vertices adjacent to that set, informally denoted the *front*. We also maintain a bounding sphere, represented by its centre and radius.

Initially, Σ contains the starting vertex, F the neighbours of that vertex, and the bounding sphere has centre at the vertex and radius 0.

We then iteratively pick, from F, the vertex closest to the centre of the bounding sphere and add it to Σ. We also remove it from F and add all neighbours not in Σ to F and then update the bounding sphere to encapsulate the vertex.

This procedure is repeated until the graph induced by F consists of more than one connected component. The process is visualised on figure 4 , 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.

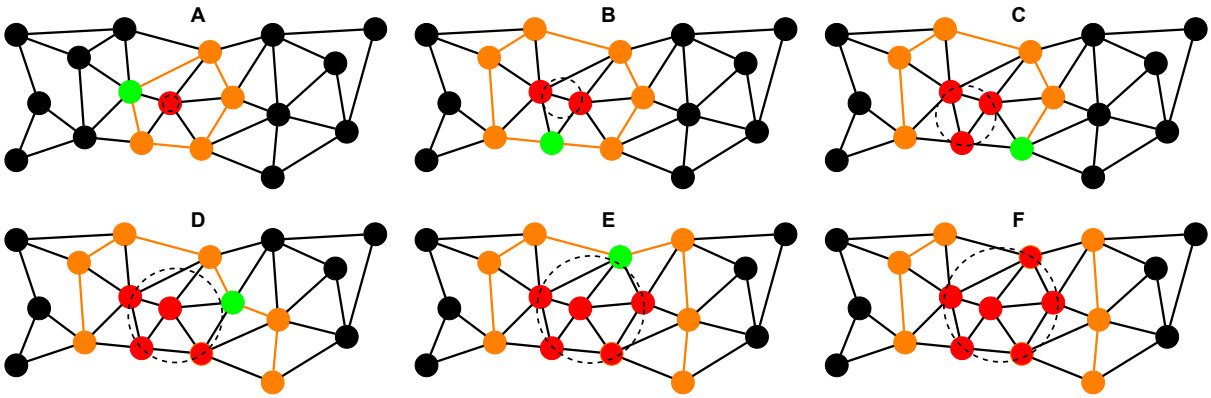


Figure 4: The process of growing a separator. Red vertices are those currently in the separator, orange vertices and edges are those currently in the front, and green vertices are the vertices of the front that are closest to the centre of the bounding sphere, which is indicated by the dotted circle. This figure is heavily inspired by figure 6 of [2].

The LSS algorithm as presented in [2] uses a linear scan in each iteration to determine the components of F. In a previous project (appendix B), we introduced a dynamic connectivity data structure[5], along with minor adjustments to the iterations of the algorithm, to explicitly maintain the graph induced by F within the structure.

Let us then briefly consider the theoretical running time of this phase of the algorithm in terms of growing a single local separator, given the conditions described in section 4.1

Since the size of the separator is Σ and we add one vertex each iteration, we use Σ iterations selecting the closest vertex from the front, updating the bounding sphere and maintaining the dynamic connectivity structure. Selecting the closest vertex is done naively with a scan through the front, taking $O(V')$ time and updating the bounding sphere takes $O(1)$ time each iteration. This gives a running time of $O(\Sigma V')$. Each edge in the dynamic connectivity structure is inserted, removed and queried for connectivity at most once, with each operation taking $O(\log^2 V')$ amortized time, totalling $O(E' \log^2 V')$.

The total running time is then $O(\Sigma V' + E' \log^2 V')$. For sake of later analysis, this running time is worth noting, but a general worst case upper bound is that the neighbourhood covers the entire graph and $\Sigma = O(V)$, resulting in a bound of $O(V^2 + E \log^2 V)$ for growing a single separator.

4.1.2 Minimizing Separators

After growing a local separator, we shrink it so that it becomes a minimal local separator. A naive way to determine a minimal local separator, given a local separator, is to simply pick a component of the front and choose the vertices adjacent to that component. The choice of component would then obviously affect what the resulting separator becomes, and it may also be a separator that does not capture the geometry of the shape as illustrated on figure 5 (B).

To better capture the geometry, the LSS-algorithm uses a heuristic with the goal of minimizing along a thin band around the shape.

The heuristic is computed by performing a Laplacian smoothing of the vertices in the separator, using inverted edge lengths as weighting scheme, and then assigning the heuristic for a vertex as the geometric distance from that vertex to the centre of the bounding sphere.

The vertices are then sorted according to their heuristic values, and we iteratively move vertices adja-

cent to exactly one front-component from the separator to that component. This is repeated until no more vertices can be removed from the separator.

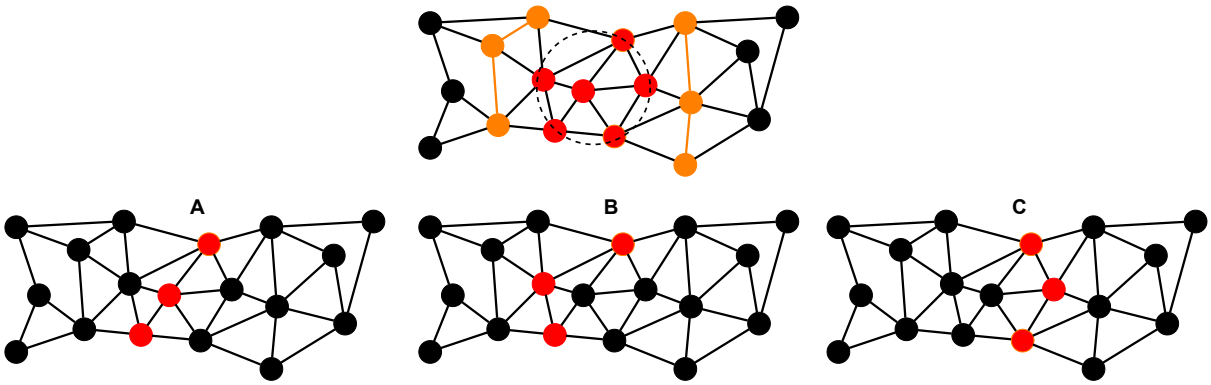


Figure 5: Minimal local separators from shrinking separator of figure 4 F. Red vertices are those currently in the separator, orange vertices and edges are those currently in the front, and the bounding sphere is indicated by the dotted circle. A shows a minimal separator that could be found by the heuristic approach, while B and C shows minimal separators found by choosing vertices adjacent to a front-component

Figure 5 shows three possible minimal separators from the separator shown on figure 4. An example of a separator that would be found using the heuristic approach is shown on (A), while (B), and (C) show the separators found by naively choosing vertices adjacent to front components. Note that (A) and (C) seem to capture the shape better than (B).

Let us then consider the time it takes to shrink a separator, Σ, to become minimal, given the conditions described in section 4.1.

To smooth the attribute of a vertex, we have to consider the positions of that vertex' neighbours, giving a time complexity on the order of the degree of the vertex. Since we do this for every vertex in the separator, the time is proportional to the sum of degrees of vertices in the separator which is then bounded by two times the number of edges in the neighbourhood of the separator, by the handshaking lemma. The time to smooth is then $O(E')$. Computing the heuristic after smoothing takes $O(1)$ time per vertex, totalling in $O(\Sigma)$ time. We then sort the vertices according to the smoothed attributes in $O(\Sigma \log \Sigma)$ time.

After sorting we iteratively move vertices from the separator to front components. In the worst case we move only a constant number of vertices in each iteration, resulting in $O(\Sigma)$ total iterations over a collection of $O(\Sigma)$ vertices. Assuming we can move a vertex from one set to another in constant time, this step takes $O(\Sigma^2)$ time.

The total time it takes is then $O(\Sigma^2 + E')$. Once again we can pessimistically bound both sets on the size of the graph s.t. $\Sigma = O(V)$ and $E' = E$ but also $E = O(V^2)$. The time it takes to shrink a single separator is then bounded by $O(V^2)$ in the worst case.

4.1.3 Separator Packing

The LSS algorithm uses a greedy weighted packing scheme to select a non-overlapping set of separators from using to extract the skeleton.

The baseline heuristic for a separator is the ratio between smallest and largest front component in

terms of their cardinality. An optional step allows for the heuristic to also incorporate a "redundancy"-measure that counts how many other separators overlap the current separator.

The separators are then sorted according to the heuristic and greedily added to the resulting set as long as there is no overlap with already chosen separators.

Let us then consider the time it takes to pack a set of separators, S.

The optional redundancy-measure takes $O(\sum_{s \in S} |s|)$ for each separator, totalling in $O(|S| \sum_{s \in S} |s|)$ time. Sorting takes $O(|S| \log |S|)$ time and packing takes $O(\sum_{s \in S} |s|)$ time. Considering then that $|S| = O(V)$ and $|s| = O(V)$, $s \in S$, we get the respective bounds of $O(V^3)$, $O(V \log V)$ and $O(V^2)$. The dominating term then depends on whether we apply this optional redundancy measure, giving a total bound of $O(V^3)$, or if we leave it out giving $O(V^2)$. Since the original algorithm uses the redundancy measure, so do our multi-scale solutions.

4.1.4 Skeleton Extraction

Given a set of non-overlapping minimal separators we now want to extract the skeleton. Since this project makes no changes to the skeleton extraction procedure of the LSS-algorithm, we will give an overview of the process with a focus on intuition rather than detailed descriptions.

Initially we maximise the minimal separators by assigning every vertex not already in a separator to its closest separator measured by geometric distance along the edges of the graph.

Each maximised separator then contributes a single vertex to the skeleton graph, with a position determined as the average position of all vertices in the separator. The vertices of the skeleton graph are then connected by edges if there are vertices in the corresponding maximised separators adjacent to each other.

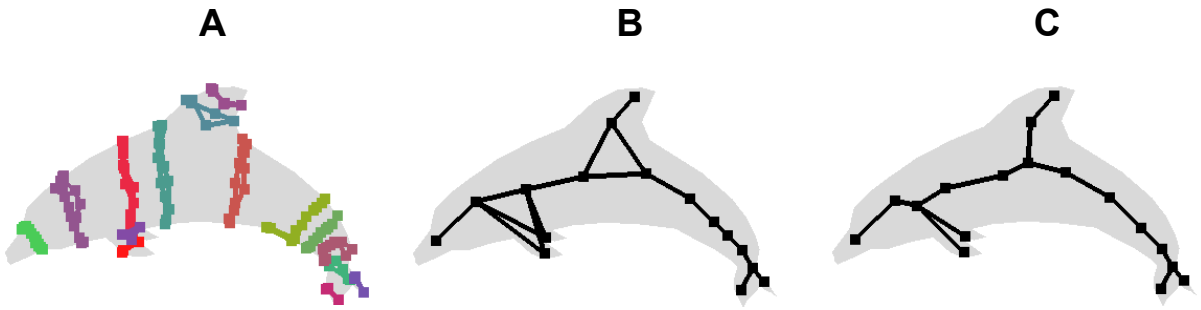


Figure 6: A visualisation of the skeleton extraction. A set of non-overlapping minimal separators (A), the skeleton graph before removal of cliques (B), and the resulting skeleton (C)

On figure 6 (B) we see such a graph obtained from the separators shown on (A). Note that this resulting graph may contain cliques of size ≥ 3 and complexes of such cliques, as seen around the fins of the dolphin. To remove cliques, we remove the edges and introduce a new vertex at the average position of the cliques, which we then connect to the vertices, giving us the final skeleton (C).

4.2 Multi-Scale Graph

We define a multi-scale graph as a collection of spatially embedded graphs, where each graph is a simplification of the same original graph. Each graph or level in a multi-scale graph has a different number of vertices but still has the same overall structure as the input graph. The result of this is that graphs with fewer vertices will also be less detailed. Thus, we can think of the resolution of a graph as

its size. We allow for lower-resolution graphs to have vertices that are not present on graphs of higher resolution graphs since this allows for graphs that better capture the structure of the input graph.

We denote each level of the input graph by an index, such that the graph M_i is the i'th level of the multi-scale graph M. Graphs are given index by their resolution such that any graph of lower index will always be of higher resolution. This means that the highest resolution graph, the graph most similar to the input graph, is denoted M_0.

The benefit of multi-scale graphs is that computations that scale with the the number of vertices run significantly faster on graphs of lower resolution. The issue with only performing the computation on very simplified version of a graph is that the loss of detail can result in insufficient results. The idea is then to run a *restricted* computation on different scales and combine the results at the end.

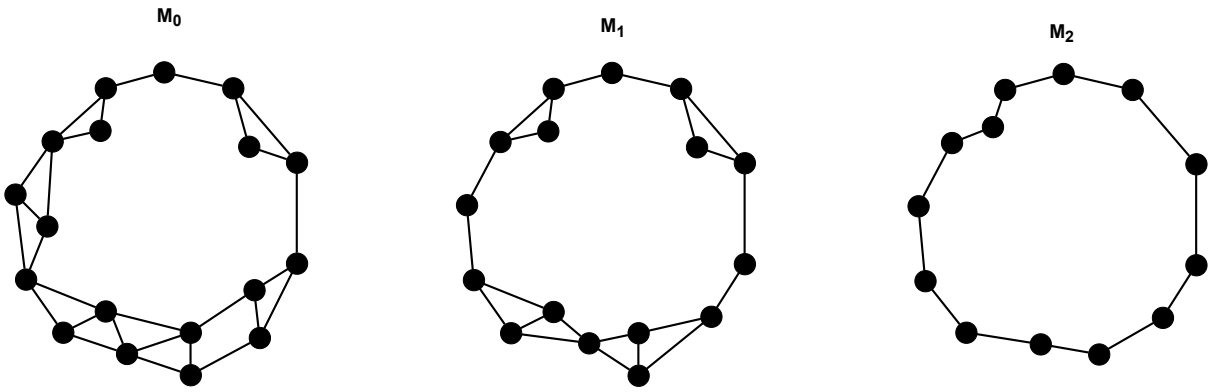


Figure 7: Example of a multi-scale graph with 3 levels.

On figure 7 an example of a multi-scale graph is shown. Note that M_0 has the highest resolution, but even as resolution decreases on M_1 and M_2, the general shape of the graph remains.

5 Testing Method/Benchmarking

5.1 Hardware and Language

We run all experiments on a Linux system running an AMD Ryzen 7 4800U CPU using all 8 cores. All implementations are written in C++17 and compiled with gcc 11.1.0 with optimisation level `-O3`.

5.2 Test Data

The models used in our experiments were provided by the authors of the original paper[2]. Of these models, **temofoam** and the the **wsv** set are voxel-grids while the remaining are meshes.

The **wsm** set and **wsv** set of models are a collection of models of the same wood statue, but each model has a different number of vertices. This is useful for determining the runtime as function of graph size for a given algorithm. We will commonly use **wsv** and **wsm** to refer to the largest models from these sets, as we use these often for different analysis as well.

5.3 Testing

Our algorithms rely on the parameter, α, which sets the threshold for maximum separator size. We will later show that choosing α is to some extent a trade off between skeleton quality and performance. Furthermore, the optimal value can also be input dependant. Since the requirements for the output can

be different depending on use-case, we are unable to select a single optimal value. Nonetheless, it is much easier to evaluate on the algorithm if the same value is used. Unless specified otherwise, we use $\alpha = 64$ in all cases. in section 8 we provide some reasoning behind selecting this value.

5.3.1 The Baseline Algorithm

To evaluate our results, it it useful to have a baseline for comparison. Because of the nature of curve skeletons, it is not possible to have a single correct skeleton to compare against. Instead we will compare against the skeletons created by LSS, which all of our algorithms are based on. Generally speaking, we can think of our solutions as trying to approximate the solution of LSS by simplifying the input. Therefore, the output will generally be of the same quality or worse. This means we can use LSS-generated skeletons as a baseline when comparing which of multiple solutions perform the best. This does mean, however, that we will end up giving positive value to some of the mistakes that LSS makes and should keep this in mind when drawing our conclusion.

The implementation we use of LSS is based on the one provided in the GEL library[6] with a slight modification to how front size ratios are calculated so that it is more in line with the description of the article[2].

5.3.2 Skeleton-quality

As we briefly outlined in the introduction, it can be difficult to decide what is a good skeleton because the properties of such can be different depending on the application. However, we still need some way of determining the overall quality of a skeleton.

Firstly, we will perform visual inspections of outputs. This is very much based on our interpretation of a good skeleton and can therefore not be used to say anything concrete, but we use this to gain intuition or guide decisions on how the algorithm should be improved.

Secondly, we will use the Hausdorff distance as a quantitative measure. In brief, the Hausdorff distance is a measure of how far two sets are from each other. The distance from a set, A, to a set, B, is measured as the largest of distances from any point in A to the closest point in B. This means that the Hausdorff distance from the A to B is not necessarily the same as the Hausdorff distance from B to A.

The Hausdorff distance can tell us how different two skeletons are from each other by measuring the distance between them. If one of the skeletons contain a feature that the other does not, this will result in larger distance between them. Since the Hausdorff distance is the greatest distance of distances, this measure will be for the greatest difference between the skeletons.

The Hausdorff distance is not symmetric, which can be used to tell if a skeleton is missing a feature or has errors when compared to a baseline skeleton. Here a large distance from the baseline to another skeleton indicates that a feature is missing from the baseline, and a large distance from the skeleton to the baseline indicates that the skeleton has found a feature that can be regarded as an error.

When we compute the Hausdorff distance, we use the available function in GEL[6]. This works by sampling a large number of points from each set and comparing the distance between them. To better visualise the data, we normalise each measurement by dividing with the bounding sphere of the used input graph.

The LSS algorithm is non-deterministic but tends to create similar outputs. When performing comparisons with a skeleton created using LSS, we will always use the first skeleton generated to avoid any bias from selecting one based on some other attribute.

5.4 Runtime testing

To measure the time it takes for an algorithm to run for a given input, we perform 3 runs and use the median runtime as our measurements. We do this to avoid outliers resulting from background processes running on the system where we run the experiments.

When we measure the runtime, we measure both the time it takes to create separators and the time it takes to perform skeleton extraction. In our project we have not changed how skeleton extraction works, and since it takes negligible time to perform, we omit it from our discussion of results.

6 Multi-Scale Approach for Skeletonization

In the original LSS algorithm, a significant amount of the computation time is spent on finding local separators. Finding large separators is especially slow since the time it takes to find a separator grows superlinear with the size of the neighbourhood of the separator. If we were to simplify the input graph before finding local separators, we should expect to find the local separators faster since they would generally be smaller. The problem with this is that the simplification will often remove details that are needed to find good local separators and in turn good skeletons.

Our solution to this problem is to only find *restricted* local separators on each level of a multi-scale graph. The restricted separators of each level are then transformed and packed into vertex sets that are also minimal local separators of the input graph. The resulting local separators can then be converted to a curve skeleton according to the LSS algorithm. This allows us to capture the general features of the input graph on low resolution levels without losing finer details that can only be found on the high resolution levels.

In this section we will show how the multi-scale graph is produced and how *restricted* local separators are found. We will also propose two primary ways of turning restricted local separators found the the multi-scale graph into local separators on the input graph. Lastly, We will introduce some potential extensions to each of the approaches that modify some part of the the procedure. For each extension of the algorithm, we will discuss our initial observation of their performance in order to inform our decisions for subsequent extensions. The result is a number different variations of the algorithm, which will be compared in section 8. To easily refer to the variations, we use **recomp**, **recomp-static**, **expand**, **expand-continuous**, and **expand-capacity** as names for the algorithms described in the sections 6.3, 6.4, 6.5, 6.6, and 6.7 respectively.

6.1 Generating Multi-Scale Graphs

To create a multi-scale graph of the input graph, we have to create a number of smaller graphs of different sizes with a similar shape. One way to achieve this is to create graph minors by performing random edge contractions on the input graph until a desired number of vertices have been removed. This will not work however, because doing so at random might contort the model in a way that does not preserve the geometric shape of the graph. If this is the case, any separators we find on these graphs will not correspond to anything meaningful on the input graph. Instead, we will use an edge contraction algorithm already present in the GEL library [6].

The benefit of using the GEL edge contraction algorithm is that it attempts to maintain the shape of the graph. It works by prioritising shorter edges first when deciding which edges to contract. Furthermore, the algorithm performs contractions in batches where each vertex can only be part of a single contraction. For each batch, the algorithm will consider the edges by order of their priority and attempt to create a maximal matching. Once all vertices have been matched or all edges have been considered,

the matched vertices are contracted and the edges are re-prioritised. This is repeated until the desired total number of contractions have been performed or no contractions are possible.

By doing work in batches, the algorithm ensures that when we perform a large amount of edges contractions, the changes to the graph will be distributed over the entire graph[7]. Also, because the shortest edges are removed first, the overall shape of the graph is less affected.

The actual edge contractions are done by picking an edge and merging the two vertices the edge connects. The resulting vertex is then moved to the average position of the two vertices.

We also have to consider the number of levels we wish to create. If the multi-scale graph does not contain enough levels, we risk missing important details of a model. At the same time, having too many levels will make our algorithm slow. Since the time to process a graph will be dependent of the number of vertices and more levels will result in more vertices. One way to balance this is to create each level such that is it half the size of the previous level. This can at most result in $log_2(n) + 1$ levels including the input graph, where n is the number of vertices in the input graph.

To create the multi-scale graph, we let the first level be the input graph. This ensures that the multi-scale graph contains at least one instance of even the smallest details. All subsequent levels are generated by iteratively contracting half the edges from the previous graph. This means that the size of M_i is $\frac{n}{2^i}$. Figure 8 shows some examples of the multi-scale graphs, where the size of each level is reduced by half.

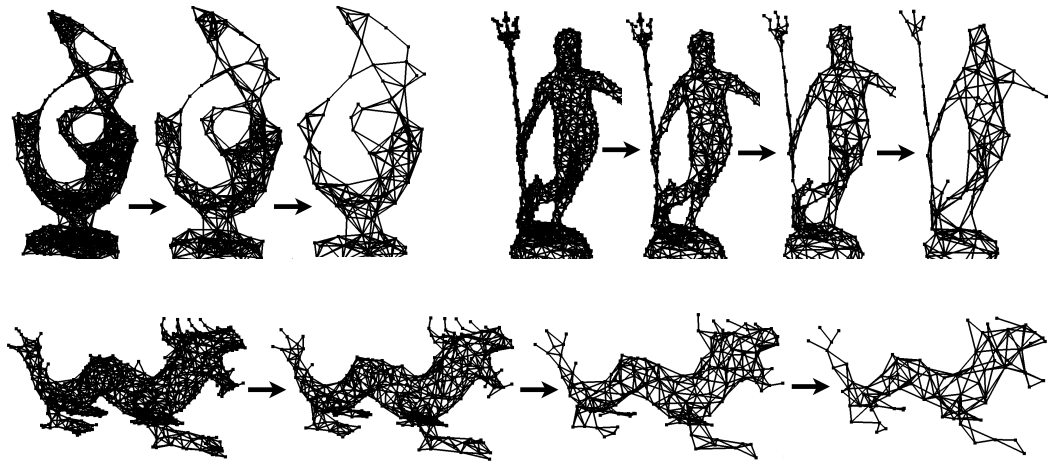


Figure 8: Shown are some of the levels that are generated when an input graph is turned into a multi-scale graph for 3 different input graphs. For each input graph we see that the level of detail decrease together with the the number vertices, while the general geometric shape of the input is still preserved. The graphs in the upper-left corner is an example of the method working on voxel-grids, while the other graphs are generated from meshes.

As will be evident later, it is useful to know which vertices were contracted together on each level. For each vertex in the multi-scale graph, we store the set of vertices on the previous level that were contracted together to create the vertex. This is kept in an array for each level with each array storing only the sets for the respective level. This effectively maps vertex-numbers on each level to a set of vertices on the previous level. We denote this mapping the "expansion-map" as it allows us to *expand* vertices to be used at a level below. The map is trivially constructed during the contraction step by keeping track of which vertices are being combined.

Another option of creating the multi-scale graph is to create each level directly from the input graph by contracting the necessary number of edges to achieve the required graph-scale. This is slightly slower

as the same edges might be removed multiple times, but it allows us to generate an expansion scheme that transforms vertices directly into the input graph.

We refer to the two options for creating multi-scale graphs as the recursive method and non-recursive method, and we will make use of both in the subsequent sections. We note that the two methods do not create exactly the same graphs, but as shown by figure 9, the results are largely similar with the non-recursive method producing slightly better results. This is likely due to how the edge contraction algorithm is used. Since it attempts to contract vertices in batches, the results will be better if all contractions are done in a single step since this means the changes are better distributed throughout the graph. These inaccuracies seems to have little effect on results and we will therefore continue with this method where it is needed.

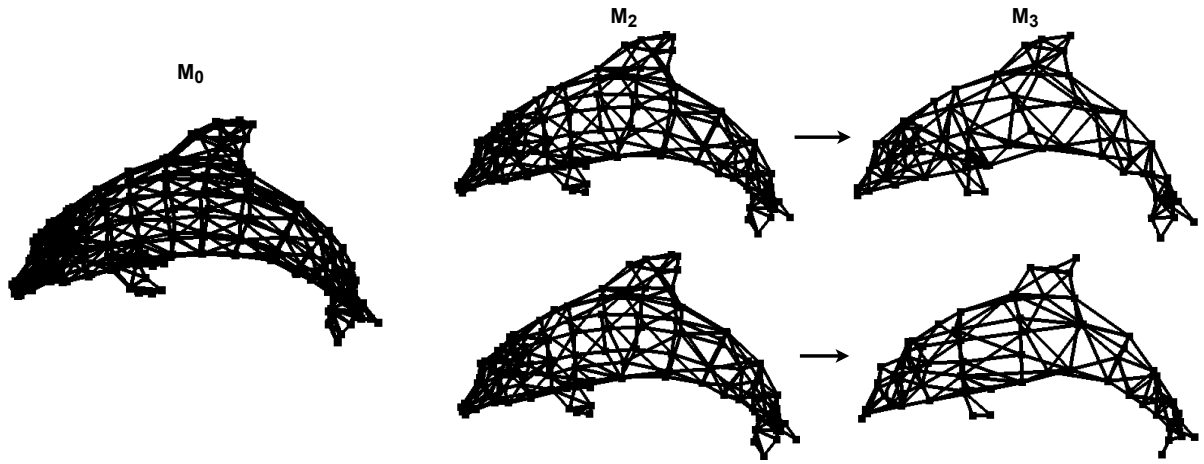


Figure 9: Comparison of the non-recursive (top) and the recursive method (bottom) for creating a multi-scale graph. Both graphs are created from the **dolphin** model. The image shows the input graph (M_0) to the left, and the M_2 and M_3 graph generated with each method to the right. We should notice that the graphs generated with the non-recursive method resembles the input graph slightly better; most noticeable with the fins of the dolphin.

6.1.1 Analysis

Given an input graph, G, of n vertices and m edges, let us consider some of the properties of the multi-scale graph, M. For sake of brevity, we will assume that G is connected. It does not strictly need to be, which can lead to cases where it is not possible to simplify the graph further before reaching $M_{\log_2 n+1}$, since every connected component has been contracted to a single vertex, however these cases make no difference for the following.

As previously noted, the number of vertices of M_i is $O(\frac{n}{2^i})$ since we halve the number of vertices every level. This also gives us a bound on the number of levels before the graph has been contracted to a single vertex, which is then $O(\log n)$. Let us then consider the number of vertices present across all levels of M as this becomes relevant for our proposed algorithms.

Clearly, the number of vertices across all levels can be described by the sum $\sum_{i=0}^{\log_2 n} \frac{n}{2^i}$, which is a geometric sum bounded by $2n$, which means the total number of vertices across all levels is $O(n)$.

Let us then consider the time complexity of building the multi-scale graph.

To build a level of the multi-scale graph, we first sort edges according to their euclidean distance. We then contract edges of a maximal matching by considering untouched edges in order of increasing

distance. If we reach the desired number of contractions, we simply stop. If the matching is not perfect, we cannot contract the desired number of edges in a single pass. Thus, we recalculate the distances and repeat, until the desired degree of simplification is reached. In the case where there are no edges, we simply stop.

As noted, we can only halve the number of vertices in a single pass if the matching is perfect, and in fact we may only contract a single edge in the worst case, as would be the case on a star graph.

Contraction can be done in constant time, so the time to build a level will be dominated by sorting edges.

For the recursive approach there are $\frac{n}{2^i}$ edges to contract to construct level i and $O(m)$ edges in the graph; the time it takes to construct M_i is $O\left(\frac{n}{2^i} m \log m\right)$. Taking construction of the entire multi-scale graph into account, we then get the bound $\sum_{i=0}^{\log_2 n} \left(\frac{2}{n^i} m \log m\right) = m \log m \sum_{i=0}^{\log_2 n} \frac{2}{n^i} = O(nm \log m)$.

For the non-recursive approach, we have to contract $n\frac{2^i-1}{2^i}$ edges since we always generate the current level by contraction on the input. This means a single level takes worst case $O(nm \log m)$ to construct. Since there are $O(\log n)$ levels, the total time becomes $O(n \log nm \log m)$.

It should be noted that edge contractions are suitable for our application since contraction is an operation that maintains the Euler-characteristic of the graph, and thus hopefully the topology[4]. In fact, the levels of the multi-scale graphs are graph minors of the input, meaning properties such as planarity remain even through simplification.

6.2 Finding Restricted Local Separators.

We stated that we wish to find *restricted* local separators on each level of the multi-scale graph. The goal here is to limit the computation time that can be used to search for a suitable separator. We recall that in the LSS algorithm, separators are grown from some initial starting vertex by adding one vertex at a time. By limiting the maximum number of vertices a separator can be grown by, we should expect that the computation would also be limited.

Let the limit for separator size be some constant ,α, if we pick α to be sufficiently low, we expect that the computation time could be lowered to a degree such that a separator is either rejected or found relatively fast. This allows us to search for restricted separators from every vertex in the multi-scale graph faster than we can find non-restricted local separators on the input graph.

The algorithm for finding restricted local separators is then as follows. From every vertex in a multi-scale graph, attempt to grow a separator in a similar manner as the LSS algorithm. If the size of the potential separator exceeds some threshold value, α, the separator is discarded. After finding a separator it is minimized as in LSS.

As with the LSS algorithm, each separator can be found in parallel to speed up computation.

It makes little sense to search for separators on graphs of size less than α, since a restricted separator, started at any vertex in a graph of size α, is able to contain every other vertex in the graph. For this reason, The multi-scale graph generation will in practice not generate levels that have size less than α, and the last level will also be exactly α in size instead of half the size of the previous level. This will not affect the analysis of the multi-scale graph since α is a constant.

6.2.1 Analysis

Let us then consider the complexity of searching for restricted separators. We will consider both the search for a single restricted separator, as well as the time to search from every vertex across the entire multi-scale graph. Let $G = \langle V, E \rangle$ be the input graph and M the multi-scale graph generated as previously described.

In section 4.1.1, we showed that growing a local separator takes $O(\Sigma V' + E' \log^2 V')$ time and in section 4.1.2 we showed that shrinking takes $O(\Sigma^2 + E')$ time, where Σ is the size of the separator and V', E' are the number of vertices and edges respectively in the subgraph induced by the neighbourhood of that separator.

For restricted separators, we are guaranteed $\Sigma \leq \alpha$, but we must also consider the size of the neighbourhood of the separator. In general, we guarantee no bound better than $V' = O(V)$ and $E' = O(E)$, but for graphs with bounded maximal degree, such as those based on voxel grids, we get some guarantees. Let Δ be the maximum degree in G, then $V' = O(\alpha\Delta)$ since every vertex from the separator adds a maximum of $O(\Delta)$ vertices to the neighbourhood, and $E' = O(\alpha\Delta^2)$ by the handshaking lemma.

Computing a restricted separator under such conditions is then $O(\alpha^2\Delta + \alpha\Delta^2 \log^2(\alpha\Delta) + \alpha^2 + \alpha\Delta^2) = O(\alpha^2\Delta + \alpha\Delta^2 \log^2(\alpha\Delta))$. Since α is constant we get $O(\Delta^2 \log^2 \Delta)$ which for constant bounded maximal degree is then $O(1)$.

However, even if the input graph has bounded degree, we may increase the degree of vertices when contracting edges as part of generating the multi-scale graph. Rather than provide theoretical guarantees that bounded degree input gives bounded degrees across the levels of M, we will empirically examine the maximum degree of multi-scale graphs generated on the models we concern ourselves with in this project.

Model	Δ_0	Δ_1	Δ_2	Δ_3	Δ_4	Δ_5	Δ_6	Δ_7	Δ_8	Δ_9	Δ_{10}	Δ_{11}	Δ_{12}
fertility	10	11	12	11	12	11	10	10	10	9	10	7	6
human_hand	31	22	16	12	12	12	11	12	9	9	9	7	-
wsv	26	42	44	39	37	33	33	23	18	13	8	5	5
wsm	11	11	12	11	11	11	12	10	11	9	8	5	-

Table 1: The maximum degree across the levels of M for various models. Here Δ_i denotes the maximal degree of M_i.

On table 1, the results of measuring the maximum degree across the levels of M can be seen for various models. Note that the maximum degree can increase from one level to the next, both for meshes and voxel-grids, as can be seen going from M_1 to M_2 on **wsv** and **wsm**. However, we observe no cases where the maximum degree of a higher level exceeds $2\Delta_0$, which implies that there is at least some semblance of a bound on the maximum degree of higher levels, on the models that we have measured.

The general worst case bound remains $O(V^2 + E \log^2 V)$ for computing a restricted separator. Considering then that we grow a separator from every vertex across every level, the total time becomes $O(V^3 + VE \log^2 V))$, since the number of vertices across every level of the graph is $O(V)$ (see section 6.1.1).

6.3 Algorithm 1: Sample Starting Vertices (recomp)

The main issue of the multi-scale graph approach for finding local separators is how the results from the low resolution graphs can be converted to results on the input graph. The first algorithm we propose does this by using the low resolution graphs to find local separators and then recomputes them on the

input graph. The intuition here is to use information gathered on a simplified version of the problem to guide our decisions on the real problem.

In the LSS algorithm, local separators are grown from a sampled set of nodes. Here vertices are sampled using a probability based on the number of times the vertex has been included in previously grown separators. This works well at limiting the total number of separators we have to grow, but we still find that the algorithm only packs a very small proportion of the generated separators. This suggests that it is wasteful to grow a separator from many of the sampled vertices, as these separators will likely be of low quality or overlap with many other separators. We can think of a restricted separator on low resolution graph as an indication that a similar, higher resolution, non-restricted separator exists on the input graph. By using this idea to sample vertices, the hope is to have a *better* sampling of vertices such that fewer separators have to be calculated while getting the same quality of results.

6.3.1 Description

The algorithm first constructs a multi-scale graph of the input graph using the non-recursive method. This will also result in a expansion-map that maps every vertex in the multi-scale graph into a set of vertices on the input graph.

A restricted separator is now grown from every vertex in the multi-scale graph. For every successfully constructed restricted local separator, we store the vertex the separator was grown from. Since it was possible to grow a restricted separator starting from each of these vertices on one of the low resolution graphs, it is likely that starting from the same positions on the input graph will yield a similar non-restricted separators.

A separator can only be grown from a vertex, but since we have no guarantee that the starting vertex of the restricted separator is present on the input graph, we have to find a new one. We can use that the multi-scale graph algorithm maintains the rough shape of the input graph and expand our starting vertex. This gives us the set of vertices on the input graph that likely contain a vertex that is close to the starting vertex on the low resolution graph. By picking the one closest to the starting vertex, we have likely found a good candidate for a starting vertex on the input graph. We do this for each of the vertices we successfully grew a restricted separator from. This gives us a collection of vertices of the input graph, which is then our sampled vertices.

We do not have a guarantee that our sampled vertices are the ones closest to those they are derived from in the multi-scale graph. Another option for picking the sampled vertices is to simply find the closest vertex of all vertices in the input graph. The drawback of this is that we would have to maintain a space-partitioning data structure like a k-d tree to efficiently find the closest vertex to any position. Doing so might be slower than examining those produced only from expansion. It is not strictly necessary to find the closest vertex since any other nearby vertex will often give roughly the same separator. For these reasons, we have gone with the simple approach of not maintaining a space-partitioning data structure.

The last step is to use the sampled vertices to compute the non-restricted local separators. The sampled vertices might contain duplicates, which should be filtered since it is expensive to grow separators. This is done by inserting the collection of vertices into a hash-set. The algorithm then proceeds as the LSS algorithm by generating a minimal local separator from each of the sampled vertices and then packing them.

6.3.2 Analysis

We consider the time complexity of generating a skeleton with **recomp** for a graph G of n vertices and m edges. The algorithm essentially consists of two disjoint steps; sampling vertices by growing restricted separators and finding and packing separators on the input graph. We first determine the time complexity for sampling vertices.

The first step in sampling vertices is generating the multi-scale graph using the non-recursive method. This was shown to be $O(n \log nm \log m)$ in section 6.1.1.

Next a restricted separator is grown from every vertex in $O(n^3 + nm \log^2 n)$ time. Let S' be the collection of successfully grown restricted separators. For each separator in S', we now find a vertex on the input graph by examining the vertex set generated from expanding a single vertex. In the worst case this takes $O(n)$ time for each separator since a vertex can be the result of contracting the entire input graph into a single vertex.

The final step of sampling is filtering the resulting vertices from the last step. This is done by inserting and extracting every element from the collection into a hash-set and takes $O(|S'|)$ time, assuming $O(1)$ operations.

Finding and packing non-restricted minimal separators from the sampled vertices is done exactly as in LSS. Since it is possible for every vertex to be part of the sampling, the worst case running time becomes $O(V^3 + VE \log^2 V)$.

Given that $|S'| = O(n)$, creating and packing of non-restricted is the dominating part and the total runtime is $O(V^3 + VE \log^2 V)$, same as running LSS with no sampling.

6.3.3 Implementation

As previously mentioned, restricted separators are grown in parallel for faster computation. Likewise, the expansion and picking of closest vertex is also an independent operation, and can therefore be done as a part of the already parallel growth step. Despite this, unlike the GEL implementation of LSS, our algorithm is deterministic because we initially grow a separator from every vertex instead of sampling based on probabilities that depend on thread scheduling.

A part of the algorithm filters duplicates from the list of sampled vertices. This is done using an `unordered_set`, which stores a unique collection of elements and uses hash values for fast access. To generate the hash values, we simply use the default C++ hash function to hash the vertex number of each sampled vertex.

6.3.4 Discussion

We run the algorithm on a small sample of models to gain an initial understanding of its performance. We first examine the skeleton output shown in figure 10. At a glance, the skeletons seem to be of a decent quality and similar to the output of LSS (appendix figure 27). Overall, the algorithm appears to capture both large and small features of the models, which indicates that the multi-scale graph works as intended.

In table 2 we compare the runtime of LSS and **recomp**. Here we find that **recomp** runs much faster than LSS. A concern about the multi-scale approach is that generating the multi-scale graph is too computationally expensive, but we see that it is not the case for any of the selected models. The little time it takes to sample vertices, by growing restricted separators and creating the mutli-scale graph, is more than made up for by the much faster time for finding and packing separators.

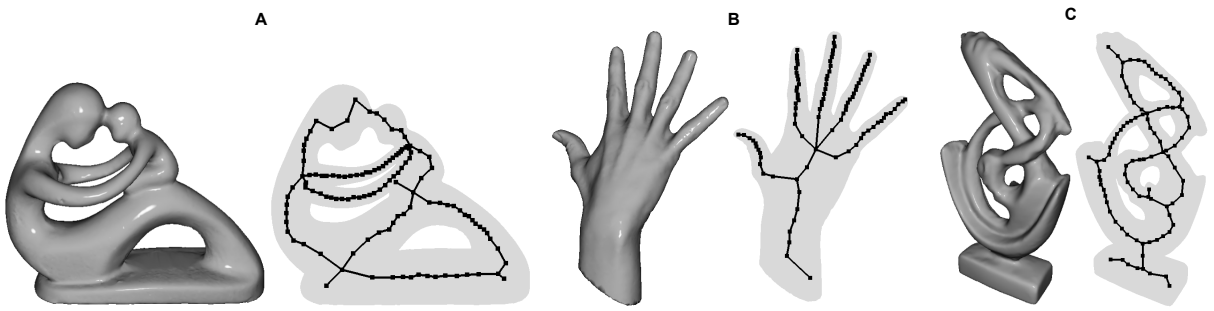


Figure 10: Skeleton output of running recomp on two meshes (A and B) and one voxel-grid (C). To the left of each skeleton is shown a shaded render of the model used to as input for the skeletonization. The models are **fertility** (A), **human_hand** (B), and **wsv** (C)

Model	fertility	human_hand	wsv
LSS			
total	171.36 s	26.82 s	137.55 s
finding separators	114.92 s	19.41 s	126.38 s
packing separators	56.36 s	7.38 s	11.07 s
recomp			
total	13.51 s	5.42 s	42.66 s
multi-scale graph	1.43 s	0.49 s	3.32 s
sample separators	2.92 s	1.49 s	11.53 s
finding separators	7.63 s	2.60 s	22.19 s
packing separators	1.46 s	0.81 s	5.47 s

Table 2: Runtime comparison of LSS and **recomp** on a small sample of models. The table shows the total runtime and the runtime of some important segments of each algorithm

To understand why finding and packing run so much faster, we examine table 3. We find that the number of computed separators, also the number of vertices that have been sampled, is much lower for **recomp** compared with LSS. We should expect the time it takes to find separators to be proportional with the number of computed separators, if the starting vertices were chosen at random. On the model **fertility**, we instead find that separators are found 15 times faster with **recomp** while 1/3 the number of separators are being computed. When we measure the average separator size for this model, we find it to be 72 with LSS and 43 with **recomp**. This indicates that the vertices we do sample will generally grow into separators faster than those sampled by LSS. Even if this is the case, we note that finding separators still seem to be the largest contributor to the runtime.

Continuing with table 2, we see that a lesser speedup is a achieved on **wsv** compared to **fertility**. To understand why this happens, it is helpful to think about how the input affects the performance of the algorithm. Take **wsv**; the model is a voxel-grid consisting of a number of relatively thin tubes. Because of its "thinness", it is very easy to successfully grow a restricted separator from most vertices, even on high resolution levels. This get amplified by the model being a voxel-grid, as separators are found faster when starting from a vertex on the inside of a shape and voxel-grids contain many such vertices. The result of this is a large proportion of the total vertices are sampled, as is also evident by the relatively high number of computed separators for **wsv** with **recomp** (table 3).

The model **temofoam** can be used to demonstrate this effect since the model is a voxel-grid consisting of nothing but thin tubes. Table 4 shows that it takes longer to skeletonize the model with **recomp** than with LSS. As theorised, the number of computed separators with **recomp** is also extremely high

Model	fertility	human_hand	wsv
LSS			
separators computed	7684	3454	2953
separators found	7668	3454	2953
separators packed	124	92	88
recomp			
restricted separators computed	49959	24845	41917
restricted separators found	2638	2018	3059
separators computed	2153	1629	2310
separators found	2153	1629	2310
separators packed	115	94	87

Table 3: Measurements from running LSS and **recomp** on a small sample of models.

for this model. One simple way to get around the problem is by creating smaller restricted separators by using a lower value for α. For this particular model, this effectively means that less restricted separators will be successfully grown on the higher resolution levels, thus fewer vertices are sampled. A simple test with **recomp** using an α value of 8, shows that less separators are sampled and that the runtime then becomes faster than LSS (table 4). Some of the decrease in runtime will come from restricted separators being faster to calculate, but we also see a much lower number of non-restricted separators being computed.

	total runtime	separators computed
LSS	60.26 s	15082
recomp	312.29 s	45455
recomp-8	40.79 s	12254

Table 4: Results of running LSS, **recomp**, and **recomp-8** (the **recomp** algorithm with $\alpha = 8$) on **temofoam**. Note that the number of separators computed is equal to the number of sampled vertices.

Using a different value for α could mean that the resulting skeleton is of lesser quality but this has not shown to be the case as seen on figure 11. However, choosing a lower value for α will not always work as it can result in skeletons missing important features of a shape. The reason it works here is because the model has roughly the same "thinness" over the whole model. It does however showcase that the issue with performance for this kind of input is due to a high sampling rate. We further explore how α affects the resulting skeleton in section 8.

6.4 Algorithm 1 Variation: Growing From Static Centre (recomp-static)

In section 6.3 we found that finding separators is still a large contributor to the total runtime of the algorithm. We propose a method for reducing runtime by utilising some of the information generated from finding restricted separators.

As part of growing both restricted and non-restricted separators, we maintain a approximate bounding sphere of the vertices in the separator. The bounding sphere is useful because we use the distance to its centre as the heuristic for deciding which vertices to add to the potential separator. If we knew the centre beforehand, we could in some cases find the separator faster because we would need to add fewer vertices before finding it. One reason for this being that the centre can be positioned inside of the shape from the beginning, allowing for an often better heuristic.

When we sample a vertex by growing a restricted separator on a low resolution graph, we can think

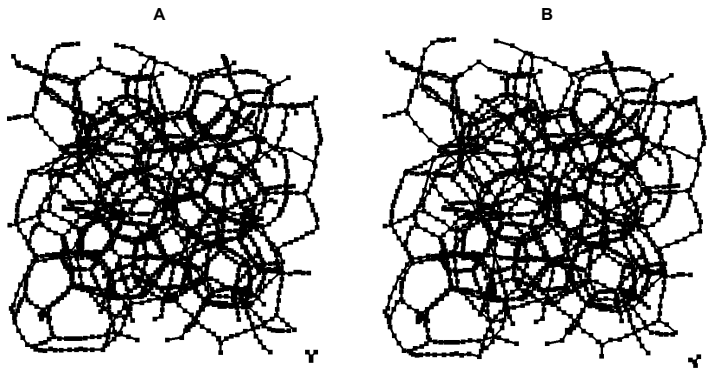


Figure 11: (A) is the skeleton from running **recomp** on **temofoam** with $\alpha = 64$. (B) is the same but with $\alpha = 8$.

of that restricted separator as simplified version of a non-restricted separator started from the same position on the input graph. Therefore, the bounding sphere that is generated as part of the restricted separator is often similar to the bounding sphere of the non-restricted separator it represents. This means we have an approximation for the centre of the final bounding sphere for every separator that we wish to generate.

6.4.1 Description

The algorithm is a simple modification of the **recomp** algorithm. For each vertex that is sampled, we also store the position of the centre of the bounding sphere that was created as part of sampling the vertex. When growing the non-restricted separators, instead of calculating distances to the centre of a dynamically changing bounding sphere, we use the position we stored together with the sampled vertex.

6.4.2 Analysis

This algorithm has the same time complexity as **recomp** since updating the bounding sphere is already a constant time operation per vertex added, and the change to the heuristic has no impact on the time it takes to calculate it.

6.4.3 Implementation

The only implementation detail to consider is how sampled vertices are filtered. Since we filter the sampled vertices in the first place, that means that some of the sampled vertices can have multiple restricted separator associated with them. It is likely that these separators have produced different bounding spheres and we have to decide if we are to include each one individually. Through some testing we have found that doing so tends to increase computation time without any noticeable gains in output quality. For these reasons, the sampled vertices are still only filtered based on their vertex number.

There is still a decision to make on which of multiple centres to include for a given sampled vertex. The current implementation simply keeps the first and discards any that follows.

6.4.4 Discussion

Since the position we use to calculate distances is an approximation of the theoretical final position of a bounding sphere, we should expect some decline in the quality of the generated skeletons. We

compare the skeletons generated between **recomp** and **recomp-static** in figure 12. Here we do see some differences between the skeletons and **recomp-static** does seem to make what could be seen as minor errors. However, while it is difficult to know for sure for such a limited set of models, the algorithm does seem to still produce skeletons of decent quality.

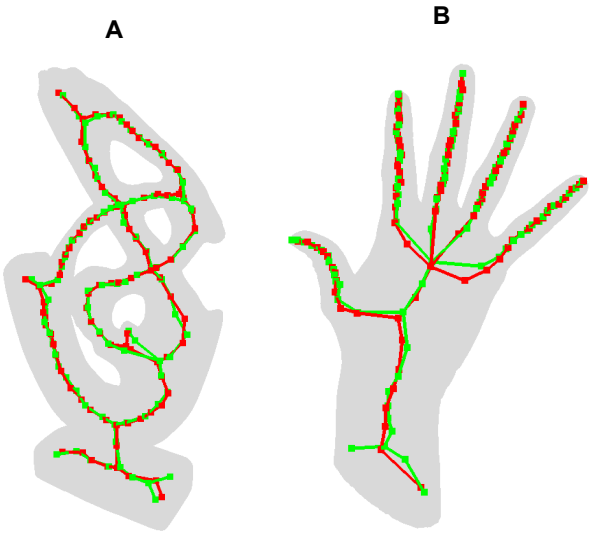


Figure 12: Skeletons of **recomp** in red and **recomp-static** in green on the models **wsv** (A) and **human_hand** (B).

Next we look at the runtime of the algorithm to see if we have gained any performance. In table 5 we show the results of running the algorithm on the same models as we used in table 2. As expected, we find that finding separators is now faster for all models. An interesting result is that we see a lesser improvement for the voxel-grid compared to the two other models. This is likely because the voxel-grid already has many vertices on the inside of the shape as opposed to the meshes.

Model	fertility	human_hand	wsv
total	8.68 s	4.64 s	35.23 s
multi-scale graph	1.45 s	0.49 s	3.31 s
sample separators	3.05 s	1.50 s	11.54 s
finding separators	2.61 s	1.85 s	15.31 s
packing separators	1.51 s	0.77 s	4.91 s

Table 5: Total and segmented runtimes of **recomp-static** on small sample of models.

6.5 Algorithm 2: Separator Expansion (expand)

An alternative way to transform a separator from a low resolution graph into one on the input, is to reverse the simplifications that resulted in the vertices of the separator at that lower resolution graph.

The idea is thus to find restricted local separators on each level, while expanding and accumulating separators from previous levels as we increase the resolution, until we have transformed every separator into one on the input.

Figure 13 shows an example of finding a minimal separator on a simplified input (C) as well as how the separator looks after expansion (D). Note that expansion can never reconnect a disconnected front, meaning a local separator stays a local separator after expansion, however it might not be minimal. To ensure that the separator is minimal we can simply shrink it again, as shown on (E).

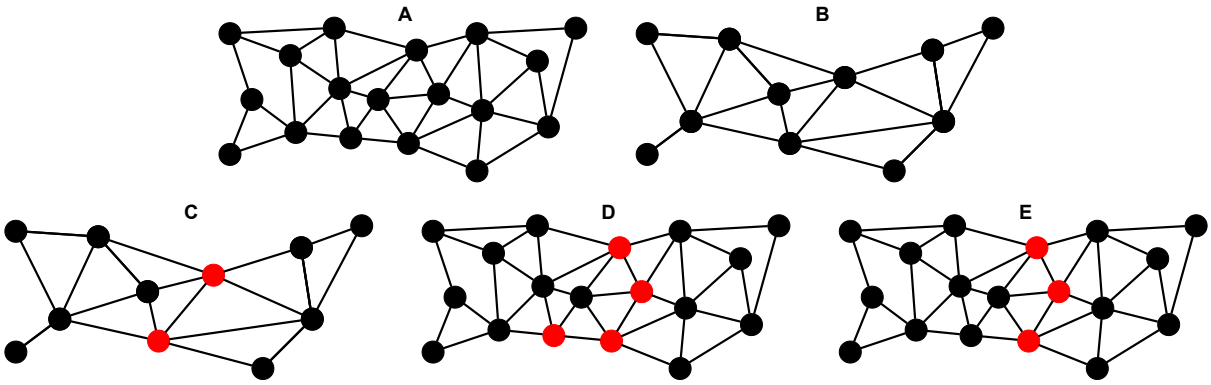


Figure 13: An input graph (A), its simplification (B) and the process of computing, expanding, and shrinking a separator (C-E).

This approach saves re-computation of separators, at the cost of expanding and shrinking. Intuitively, we also now find the actual separators from lower resolutions, as they are represented on the input, in contrast to finding separators that are grown from similar starting positions.

This then should allow us to capture small features at high resolutions, and as resolution decreases the separators we find and expand will represent features of increasing size on the input.

6.5.1 Description

Initially we generate the multi-scale graph using the recursive method. In doing so, we maintain a map from vertices of M_i to M_{i-1} for $i > 0$ to track which vertices they correspond to at higher resolutions. This allows for easy expansion of separators later.

We then iteratively, beginning at the highest level of the multi-scale graph, compute restricted separators on the current level, expand every accumulated separator to the next level and shrink them.

As the resolution decreases, the probability that separators grown from similar starting vertices grow into the same separator increases. In order to avoid duplicate separators, the collection of separators is filtered by inserting them into a hash-set followed by an extraction of each element from the set. This is done on every level prior to expansion. Details regarding the hashing can be found in section 6.5.3.

We use the map stored during creation of the multi-scale graph to expand separators, by unioning the corresponding vertices on the previous level resulting from querying every vertex of a separator.

In order to shrink separators using the approach detailed in section 4.1.2 we need to know the front-components of a separator. To avoid accumulating and having to expand the vertices of the front-components as separators are repeatedly expanded and shrunk, we simply perform a scan of the separator to find the front-components after expansion, allowing the same method to be used.

When the input graph is reached, we compute the last set of restricted local separators and perform packing of the accumulated minimal separators that have been transformed from every level of the multi-scale graph into separators on the input.

After packing, we simply extract the skeleton as in section 4.1.4.

6.5.2 Analysis

Let us then consider the time complexity of this approach on a graph G of n vertices and m edges. As before, we must generate the multi-scale graph and search for restricted local separators. Analysis

of these steps is described in section 6.1.1 and 6.2.1, where we find that they take $O(nm \log m)$ and $O(n^3 + nm \log^2 n)$ time respectively, totalling in $O(n^3 + nm \log^2 n)$ time for these steps.

We then have to consider the time it takes to expand and shrink separators across the levels of the multi-scale graph. We can bound the time it takes to expand, recompute the front, and compute a hash value for filtering for a single separator on level i as $O\left(\frac{n}{2^i}\right)$ since each of these procedures is linear in the size of the separator. For a single separator across all levels the time spent on these procedures is then $O\left(\sum_{i=0}^{\log_2 n} \frac{n}{2^i}\right) = O(n)$ by the fact that the sum is geometric.

In addition, we also shrink the separator on each level. Shrinking on a single level takes $O\left(\left(\frac{n}{2^i}\right)^2\right)$ time (see section 4.1.2), meaning for a single separator across every level we get the total time shrinking is $O\left(\sum_{i=0}^{\log_2 n} \left(\frac{n}{2^i}\right)^2\right) = O(n^2)$ since the sum converges to $\frac{n^2}{3}$.

The total time spent transforming a separator from the multi-scale graph into one on the input is then bounded by $O(n^2)$ and since there are $O(n)$ separators the total time spent is $O(n^3)$.

Thus, even with expansion we perform no worse asymptotically than LSS.

6.5.3 Implementation

In the same way we grow restricted separators in parallel, we are also able expand and shrink them. As we need to expand and shrink every separator on each level, this is done in separate step.

It is necessary for good performance that the accumulated separators are filtered for duplicates. However, filtering needs to be efficient since it done often. Therefore it is important that we use a hash function that results in few collisions while also being fast to calculate.

We use a simple approach of hashing the vertex number of each vertex in the separator, and combining the hashes using XOR-operations. We also need an efficient way to check if two separators are equivalent as it is also needed to maintain a hash-set. For this we use the fact that the vertex number are stored in order to check for equivalent in a single parse. Through testing we have found that this works well, as filtering generally compose a small proportion of the total runtime.

6.5.4 Discussion

We run the algorithm on a small sample of models to gain an initial understanding of its performance. We first examine the skeleton output as shown on figure 14. At a glance, the skeletons seem to be of a decent quality and similar to the output of LSS (appendix figure 27), with some minor differences. Specifically on (A) we are missing a branch of the skeleton on the bottom left, while on (B) we have an additional branch on the bottom left.

In table 6 we show the running times of LSS and **expand** on an array of varying inputs. Specifically we are interested in examining not just the total time for the multi-scale approach, but also how the various phases of the algorithm performs, to get an idea of where to improve practical performance.

We consider the time spent generating the multi-scale graph, the time spent computing restricted separators, the time spent expanding separators, the time spent shrinking separators as well as filtering and packing. What we generally see is that the algorithm performs much faster than LSS in practice, except for the case of **temofoam**. When examining what phases dominate the running time we generally see that finding restricted separators and shrinking make up a large portion of the time, but on **temofoam** packing is by far the most time consuming.

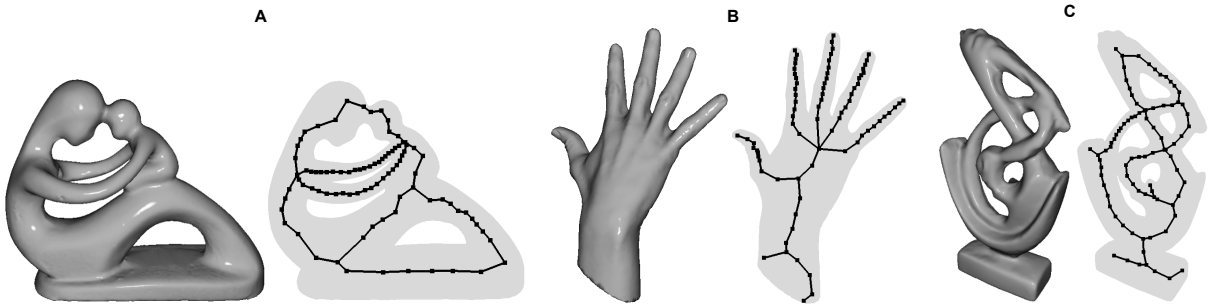


Figure 14: Skeleton output of running **expand** on two meshes (A and B) and one voxel-grid (C). To the left of each skeleton is shown a shaded render of the model used as input for the skeletonization. The models are **fertility** (A), **human_hand** (B), and **wsv** (C)

Model	fertility	human_hand	wsv	temofoam
LSS				
total	171.36 s	26.82 s	137.55 s	59.25 s
finding separators	114.92 s	19.41 s	126.38 s	27.94 s
packing separators	56.36 s	7.38 s	11.07 s	30.99 s
expand				
total	4.26 s	2.24 s	14.68 s	109.14 s
multi-scale graph	0.12 s	0.49 s	0.74 s	1.53 s
restricted separators	3.09 s	1.51 s	11.41 s	23.44 s
expansion	0.04 s	0.03 s	0.04 s	0.20 s
shrinking	1.87 s	1.11 s	3.72 s	45.10 s
filtering	0.03 s	0.03 s	0.08 s	0.27 s
packing	0.23 s	0.21 s	0.58 s	75.77 s

Table 6: Runtime comparison of LSS and **expand** on a small sample of models. The table shows the total runtime and the runtime of some important segments of each algorithm

Model	fertility	human_hand	wsv	temofoam
LSS				
separators computed	7684	3454	2953	15129
separators found	7668	3454	2953	15104
separators packed	124	92	88	2194
expand				
restricted separators computed	49917	24827	41912	115657
restricted separators found	2589	1925	3046	80734
separators packed	84	75	75	2305

Table 7: Measurements from running LSS and **expand** on a small sample of models.

Examining table 7 can give us an idea of why this is. As noted before, **temofoam** is a very thin tube-like shape, that generates an enormous amount of separators compared to the other inputs. Since the approach works by accumulating separators across levels, it makes sense that we must perform more work on shrinking and packing as the accumulated number of separators grows. We will examine how a variation of the algorithm may circumvent this issue.

6.6 Algorithm 2 Variation: Continuous Packing (expand-continuous)

The previous approach to expansion accumulates separators across the levels of the multi-scale graph before packing them all at once when arriving at the input. Shrinking generally makes up a large portion of time, and on **temofoam** packing is dominating.

The motivation behind this variation is thus to reduce the number of separators to be processed and packed at the final level, by packing on every level. Since packing removes overlapping separators, we should spend less time processing separators that will eventually be discarded. This is similar to how we shrink every level, in that we hope a reduced input to these procedures makes up for the fact that we have to apply them repeatedly.

6.6.1 Description

Recall that **expand** works by computing restricted separators on a multi-scale graph and then expanding them into separators on the input.

Previously we expanded separators from one level to another, shrunk the separators to ensure that they were minimal and performed filtering to discard duplicate separators. In addition to this, we now perform packing on each level, which in practice means a number of the separators are discarded before they reach the input graph.

6.6.2 Analysis

Packing on a graph of n vertices takes $O(n^2)$ or $O(n^3)$ time depending on if the optional redundancy measure is applied (see section 4.1.3). In either case, the algorithm performs packing on every level, where level i has $O\left(\frac{n}{2^i}\right)$ vertices. Thus we can describe the time to pack as the sums $\sum_{i=0}^{\log_2 n}\left(\frac{n}{2^i}\right)^2$ or $\sum_{i=0}^{\log_2 n}\left(\frac{n}{2^i}\right)^3$ depending on our choice of using the redundancy measure. Note that the sums converge to $\frac{n^2}{3}$ and $\frac{n^3}{7}$ respectively, meaning the total time to pack across all levels remains $O(n^2)$ or $O(n^3)$ as it was when packing only at the end.

Although we have shown no improvement to the asymptotic complexity, the goal is to see a reduction in the time spent in practice.

6.6.3 Discussion

On table 8 we show the running times of LSS and **expand-continuous** on the same set of inputs as before, with measurements of the phases we have changed. The motivation behind this approach was to reduce the number of separators to shrink and pack, and we see a very clear difference in the time spent on these procedures across every model. For **temofoam** the impact on the total running time is significant, but the improvement is otherwise minor.

We also consider the number of separators that have been packed. We expect no difference in the number of computed separators between **expand** and **expand-continuous** as we have made no changes there, however we see a very clear reduction in the number of packed separators.

On figure 15 we show the differences between skeletons obtained by **expand** (red skeleton) and **expand-continuous** (green skeleton). On (A) the differences are subtle and apart from missing the leftmost bump the skeletons are mostly similar. On (B) however, the difference in quality is more striking. Apart from the bottom left feature, which is also missing on the LSS skeleton, the **expand-continuous** skeleton also misses a separator in the middle of the hand, leading to degraded quality.

Model	fertility	human_hand	wsv	temofoam
expand				
expansion	0.04 s	0.03 s	0.04 s	0.20 s
shrinking	1.87 s	1.11 s	3.72 s	45.10 s
filtering	0.03 s	0.03 s	0.08 s	0.27 s
packing	0.23 s	0.21 s	0.58 s	75.77 s
separators packed	84	75	75	2305
expand-continuous				
expansion	0.005 s	0.003 s	0.005 s	0.027 s
shrinking	0.18 s	0.09 s	0.34 s	2.03 s
filtering	0.006 s	0.004 s	0.013 s	0.206 s
packing	0.05 s	0.05 s	0.09 s	24.26 s
separators packed	60	53	59	2169

Table 8: Comparison of **expand** and **expand-continuous** on a small sample of models. The table shows runtimes of some important segments of each algorithm, as well as the number of separators packed

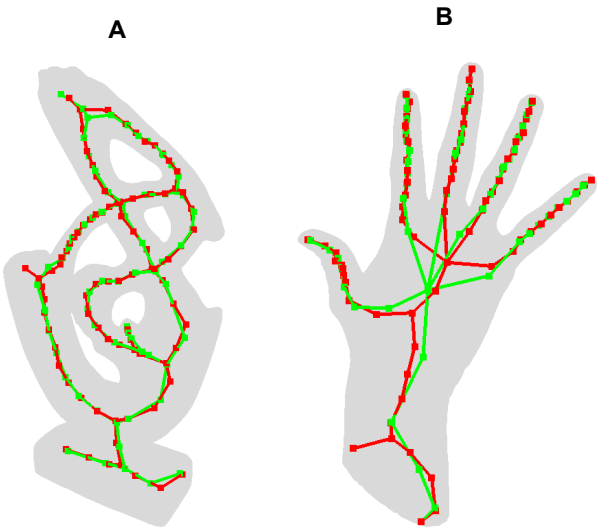


Figure 15: Skeletons of of **expand** in red and **expand-continuous** in green on the models **wsv** (A) and **human_hand** (B).

This is almost certainly because we are too eager to discard overlapping separators, so that once we arrive at the final skeleton we are missing separators needed to give the expected quality.

6.7 Algorithm 2 Variation: Capacity Packing (expand-capacity)

Although we were able to reduce the time spent on packing by the previous approach, it came at the cost of the quality of the skeletons.

When packing every level, we may discard separators that currently overlap, but may not have overlapped, had they been expanded all the way to the input graph. On figure 16 such a scenario is shown. Two separators that overlap on a high level, that cease to overlap once they have been expanded and minimised because the overlapping vertex was split into two.

To avoid discarding too eagerly, we can modify packing to only discard separators if we know there is

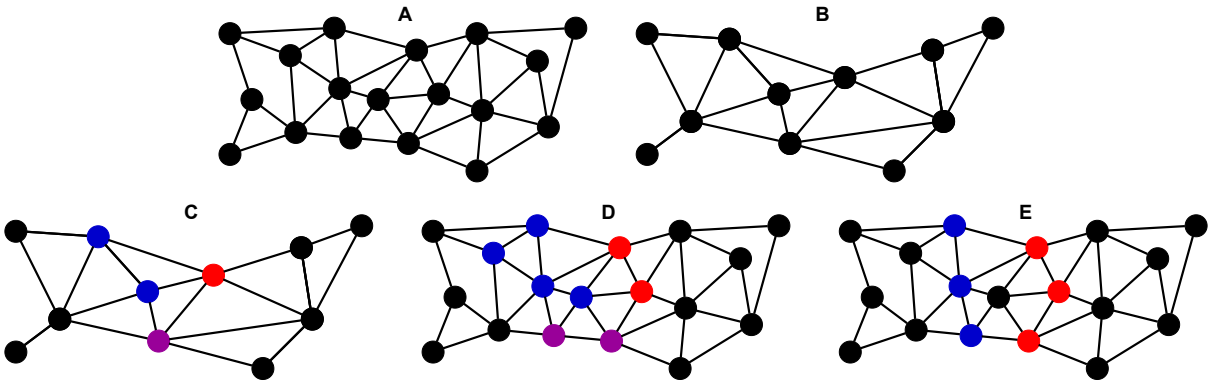


Figure 16: An input graph (A), its simplification (B), and two separators (red and blue vertices) that overlap (purple vertices) before expansion and shrinking (C-E)

overlap even after expanding to the input level. The idea is that such an approach will give us the time benefits of packing continuously, while also maintaining the quality from only packing once.

6.7.1 Description

We associate with each vertex a capacity equal to the number of vertices of the input it represents. In addition to maintaining an expansion map when generating the multi-scale graph, we also maintain a vector of capacities that we then pass along to the packing procedure.

When packing we then no longer discard a separator if a vertex is already in use, but rather if the capacity of that vector has already been reached.

6.7.2 Analysis

The difference between this variation and the previous is the changes made to packing, as well as the introduction and maintenance of capacities.

Intuitively the change to packing makes no difference for the time complexity, since the worst case remains that we have to iterate over every element of every separator (see section 4.1.3).

Tracking capacities can be done during generation of the multi-scale graph. For every contraction done, we simply add the capacities of the respective vertices to the resulting vertex. This is a constant time addition to an already constant time operation, which leaves the time complexity unchanged.

In general these changes make no difference for the asymptotic bounds, however we expect there to be difference in practice since we are now more hesitant to discard separators before reaching the input graph.

6.7.3 Discussion

On table 9 we show the running time of select phases of **expand-continuous** and **expand-capacity**. As expected, we see some increase in expansion, shrinking, filtering, and packing arising from the decreased number of discarded separators when compared to **expand-continuous**. However, the time spent is still less than that of **expand**. In addition, when comparing the number of separators packed we find that we are closer to **expand** than **expand-continuous** managed.

Examining the skeletons visually, as seen on figure 17, we see that the skeleton generated by **expand-capacity** (shown in green) more closely resembles the skeleton generated by **expand** (shown in red)

Model	fertility	human_hand	wsv	temofoam
expand-continuous				
expansion	0.005 s	0.003 s	0.005 s	0.027 s
shrinking	0.18 s	0.09 s	0.34 s	2.03 s
filtering	0.006 s	0.004 s	0.013 s	0.206 s
packing	0.05 s	0.05 s	0.09 s	24.26 s
separators packed	60	53	59	2169
expand-capacity				
expansion	0.016 s	0.009 s	0.017 s	0.063 s
shrinking	0.69 s	0.38 s	1.32 s	5.64 s
filtering	0.017 s	0.009 s	0.031 s	0.233 s
packing	0.23 s	0.24 s	0.38 s	32.13 s
separators packed	72	67	68	2218

Table 9: Runtime comparison of **expand** and **expand-capacity** on a small sample of models. The table shows the runtime of some important segments of each algorithm and the number of separators packed

than that generated by **expand-continuous**. This is most notable on (B) where the previously missing separators in the middle of the hand have been included, leading to a higher quality. Also note that the missing branch of the skeleton on (B) makes **expand-capacity** more closely resemble the skeleton of LSS.

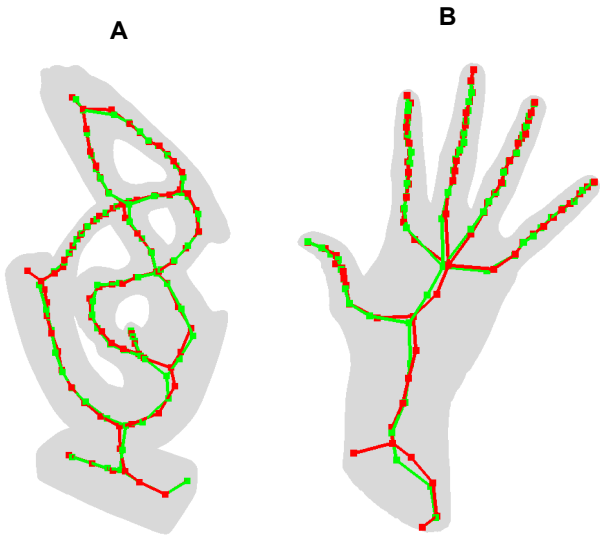


Figure 17: Skeletons of of **expand** in red and **expand-capacity** in green on the models **wsv** (A) and **human_hand** (B).

In general we find that **expand-capacity** serves as a great middle ground between **expand** and **expand-continuous**, sacrificing some number of separators packed for a vast increase in performance on models such as **temofoam** where the thin and tube-like nature causes a vast number of restricted separators to be found.

7 Potential Improvements

I this sections we propose some potential improvement to the algorithms. These are ideas that could possibly improve either the performance or the output quality for all algorithms that have been presented. For each potential improvement, we will briefly explore its validity and discuss how it could be used in each algorithm.

7.1 Multi-Scale Graph Level Count

When deciding on how to create the multi-scale graph, we picked the simple strategy of halving the problem size with each level. A worry about doing so is that we contract the graph so much in a single iteration that a feature is removed before its shape can be "found" by a restricted separator on the level before it was removed. If this is the case, having more levels in the multi-scale graph should produce better skeletons because there are smaller differences between each level. A drawback of doing so is that the multi-scale graph now contains more vertices from which to grow a separator from, increasing the runtime.

To show how using adding more levels to multi-scale graph can affect the output skeleton, we modify the **expand** algorithm to generate a different multi-scale graph. The difference between the the original **expand** algorithm and the modified version is that instead of reducing the size of each level by half, we reduce it by one fourth. In figure 18 we show the skeleton of running the modified algorithm against the skeleton of **expand**.

For the model **human_hand**, we do see some differences between the skeletons, especially in the bottom of the shape. However, it is difficult to say if the new skeleton is "better" as it is unclear what the skeleton should look like for the area of interest. On the model **wsv**, we might be able to say that the new skeleton is slightly better but it does not include any of the features that is missed by the original so the improvement is very minuscule at most. Thus, we are unable that say with confidence that having more levels in the multi-scale graphs will improve the output.

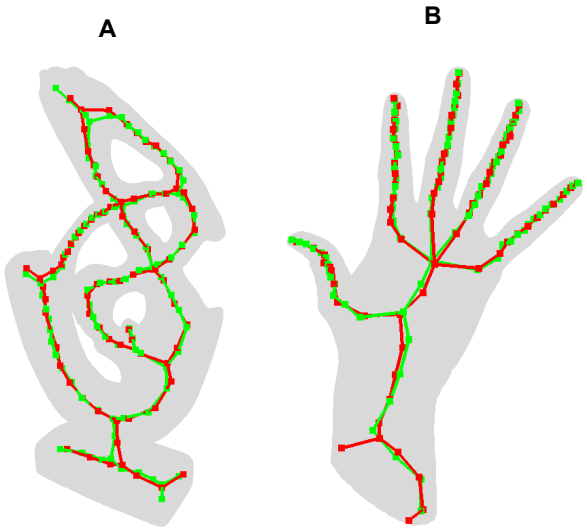


Figure 18: Skeletons generated on **wsv** (A) and **human_hand** (B) with **expand** using two different multi-scale graphs. The skeleton in red used the original multi-scale graph and the skeleton in green used a modified multi-scale graph with more levels.

When we look at the runtime of the tested models, we also find that having more levels is significantly

slower. We might be able to able lower the α value slightly to compensate, but doing so is often not worth it as it is more expensive to grow a lot of additional restricted separators than to use a higher α.

While this should not rule out that changing how the multi-scale graphs are generated could be advantageous, it does indicate that this is not the right approach. We should also keep in mind that we only showed this for two models and it could work better on others. In practice however, we find that features not being included due to being removed by contraction, only happens when we use very low values for α. The main reason for why features are not always included seems to be elsewhere but it still related to α. We will explore this further in sections 8.

7.2 Vertex Sampling

The algorithms that we have presented all grow a restricted separator from every vertex. We do this because we can calculate restricted separators relatively fast and because we want to ensure that we do not miss a feature of a shape. However, this is still a lot of computation that could possibly be avoided. Because the multi-scale graph is multiple levels of simplification of the same data, we have a lot of redundancy in our computed separators. This is especially true when we run a high α value as this means we find separators in the same space on multiple levels.

Consider a single level of the multi-scale graph. As is often the case, starting from a vertex that is already part of another separator, we often end up with a separator that is the same or very similar. This is the intuition behind the sampling scheme of LSS, where vertices are sampled based how many separators they are already part of. We can extend this idea to also work across levels such that we not only reduce the probability of growing a separator from a vertex if that vertex is included in a separator grown on this level, but also if it is part of any separator that was found on a lower resolution.

This is most intuitive on **expand** and its variations, where we can easily determine if a separator from a higher level includes a given vertex, but is not so easily adapted for **recomp** and its variation where the relations between levels are not so clear.

For **expand** and its variations, we have implemented such a sampling scheme, based on that of LSS (see section 4.1.1), where the probability of growing a separator from a vertex is 2^{-x} where x is the number of separators that include that vertex, both for separators grown at the current level or expanded to this level from lower resolutions. The results can be seen on figure 19 for **expand** on **temofoam** as a function of α. As expected, sampling makes less of a difference for smaller α but as α increases, and thus the number of redundant separators increase, we see a very drastic performance difference using sampling.

Sampling has proven to be a valuable way of increasing performance, especially since there is a large number of redundant separators grown on lower resolutions which we see most clearly on **temofoam**. Although it is not clear how adapting sampling across levels for **recomp** and its variations should be done, we can still perform sampling internally for each level. That is, we use the same sampling scheme as from LSS, but only considering separators that have been grown on the same level. Note that this sampling is separate to growing restricted separators.

On figure 20, results are shown for runtime measurements of **recomp** with and without sampling on **temofoam** as a function of α. Although there is a clear improvement, it is not competitive with the running time achieved with the sampling scheme introduced for **expand** and its variations, or even LSS. We believe some implementation of cross-level sampling is necessary, for **recomp** to run efficiently for inputs like **temofoam**, as this should greatly reduce the amount of successfully grown separators and therefore the number of sampled vertices on the input for the next step of the algorithm.

In addition to considering the computational performance when introducing sampling we must also

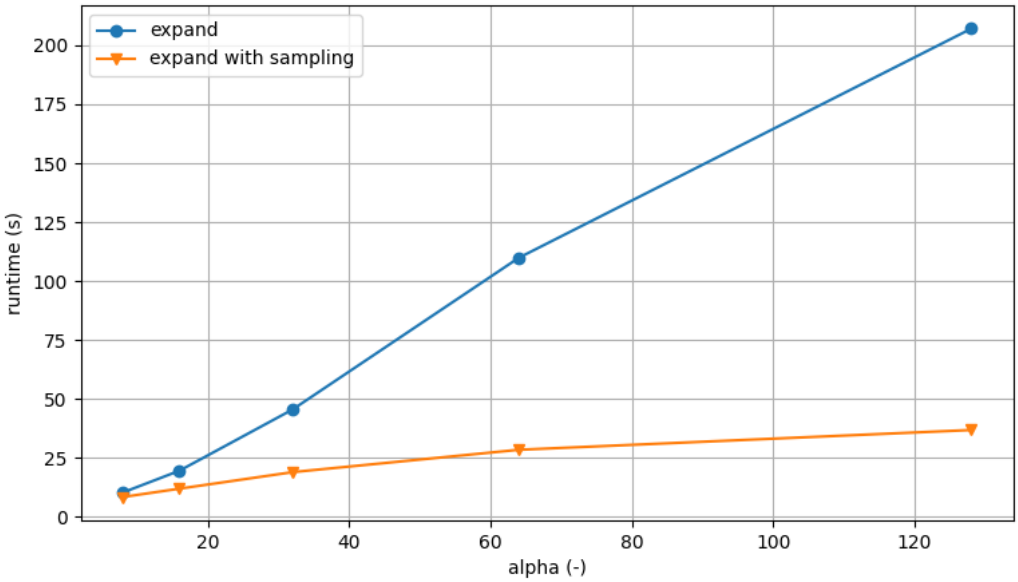


Figure 19: Runtime of **expand** as a function of α with and without sampling.

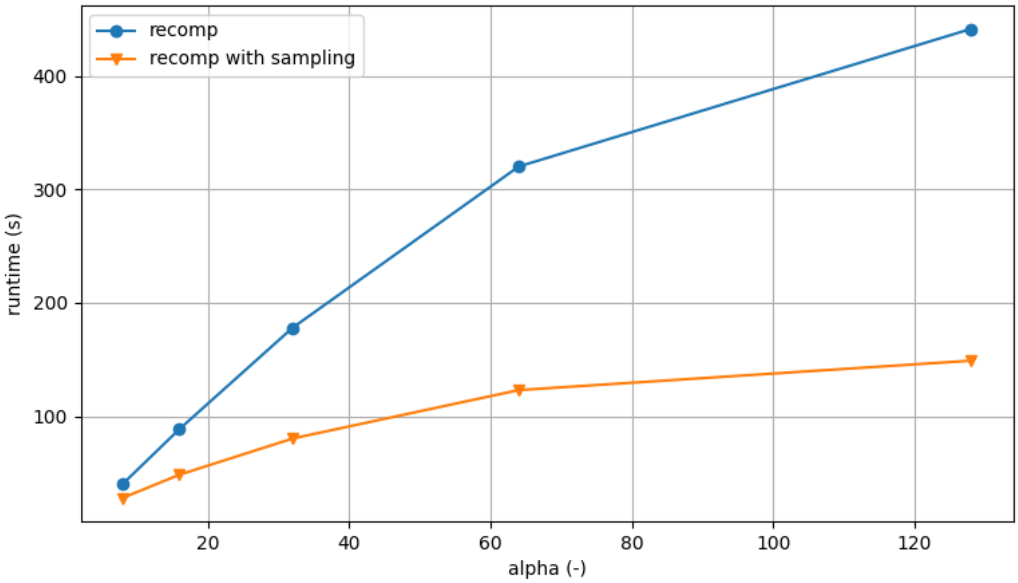


Figure 20: Runtime of **recomp** as a function of α with and without sampling.

consider the impact on output quality. On figure 21, we show skeletons obtained from **expand-capacity** with sampling (red) and without (green). With the introduction of sampling one would intuitively expect a drop in performance, since we may now miss a great separator because we did not choose to grow one from a valuable starting vertex. Some differences can be observed, notably on (B) where the bottom left branch is missed. This particular branch was also discussed in section 6.7 as it is not intuitively clear if it should be part of an ideal skeleton.

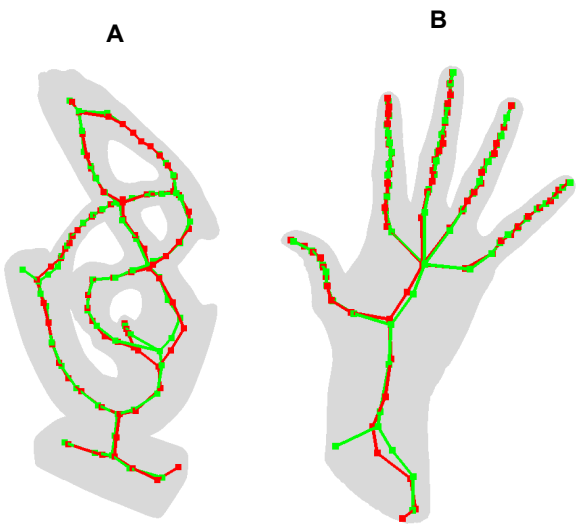


Figure 21: Skeletons of **expand-capacity** with sampling in red and **expand-capacity** without sampling in green on the models **wsv** (A) and **human_hand** (B).

In general it does seem like the performance gains achieved by introducing sampling make up for the difference in output, since it is otherwise minor.

8 Results and Discussion

It should be evident that there is a relation between the number of separators found and the value of α. If α is equal to n, it is possible for separators to grow into the entire graph, which is similar to how local separators were found in the original LSS algorithm. If α is 1, it is possible to find no separators at all since only leaf-vertices would qualify.

Having a lot of local separators makes it easier so build a good skeleton, but it also demands more computation at later stages. This means that we might have to balance quality and computation time. Let us first consider the runtime as a function of α.

In appendix, figure 33, we show that the runtime appear to grow linearly as a function of α. This seems to be the case regardless of algorithm but **recomp-static** does have a large jump in runtime for $\alpha = 256$ on **wsv**. The results are better than expected but shows that using a high α value comes at the cost of computation time.

To determine the quality of the generated skeletons, we will compare each with a baseline skeleton of the same model using the Hausdorff distance as outlined in section 5.3.2. Figure 28-32 in appendix show the Hausdorff distance as a function of α for running each algorimth on a variety of models. In figure 22 we highlight the results for the expand algorithm. It can be difficult to interpret this data as the distance seems to fluctuate, but we find that, generally, the distance from the newly generated graphs to the baseline tend decrease as α increases until a certain point where the distance become more stable. In the opposite direction, the distance seems to be less affected by α. We can interpret this to mean that skeletons generated with higher α values contain less error but only up to a certain point. This make sense since we will be able to find more separators on any given resolution level as α increases, but at a certain point, restricted separators can grow so large that we will simply be able to find most of the features directly on the input graph at level zero.

For most of our test data, we find this point to be somewhere around 50 to 100 for all of our algorithms.

Since the runtime grows at least linearly with α, we should not set α above this range. To keep things simple, we will use the same α for the the remaining experiments of this section as we have found this value to be good trade off between quality and performance. This should not be seen as a final decision for the size of α since the optimal value is highly input dependant and we have only tested for a small sample of inputs.

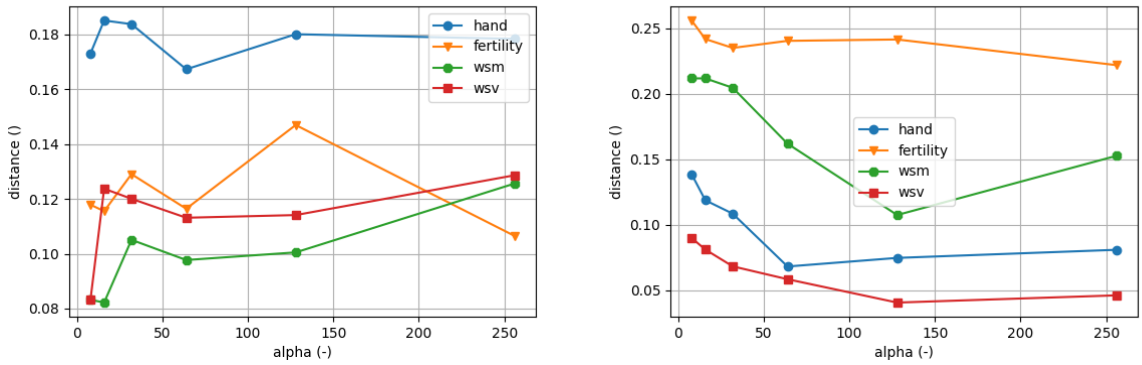


Figure 22: Hausdorff distance as a function of α for **expand**. To the left, the distance from baseline skeletons to a skeleton of the same model generated with the algorithm. The right is the reverse.

Now that that we have picked a standard value for α, we are able to examine the runtime of each algorithm as a function of the graph size. On figure 23, these results are shown. For **wsm**, the runtime of every algorithm seems close to linear which is much better than the theoretical bound suggests. It should be noted that our analysis' are of course based on worst cases, and these results are only for one shape. For **wsv** it is more clear that **recomp** and **recomp-static** are super-linear as a function of the number of vertices, however **expand** and its variants still seem close to linear.

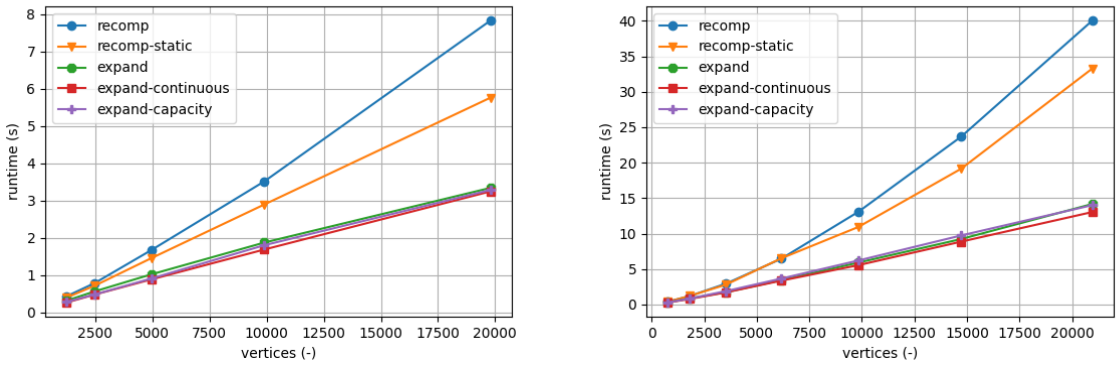


Figure 23: Runtime as a function of graph size. Plot on the left shows runtimes on the **wsm** set of models and plot on the right shows runtimes on the **wsv** set of models

For the **wsv** set and the **wsm** set, **recomp** is the slowest of our algorithms, with **recomp-static** performing better but similarly, as previous results have suggested. It should not be surprising that **expand** and its variations are very close in performance, since the variations were built around improving performance on certain types of input in the case of **expand-continuous**, while **expand-capacity** aimed to improve output quality while maintaining performance.

To determine the quality of each algorithm, we have measured the output of skeletonization against a baseline using the Hausdorff distance. In table 10, we show measurements for a variety of models.

	fertility	human_hand	temofoam	wsv	draco
LSS	(0.000, 0.000)	(0.000, 0.000)	(0.000, 0.000)	(0.000, 0.000)	(0.000, 0.000)
recomp	(0.103, 0.188)	(0.055, 0.053)	(0.028, 0.033)	(0.116, 0.052)	(0.059, 0.140)
*-static	(0.113, 0.204)	(0.130, 0.081)	(0.046, 0.049)	(0.134, 0.073)	(0.134, 0.133)
expand	(0.105, 0.234)	(0.167, 0.074)	(0.067, 0.037)	(0.133, 0.061)	(0.059, 0.139)
*-continuous	(0.124, 0.238)	(0.155, 0.141)	(0.077, 0.039)	(0.155, 0.062)	(0.080, 0.145)
*-capacity	(0.109, 0.226)	(0.155, 0.050)	(0.060, 0.057)	(0.147, 0.078)	(0.077, 0.135)
*-capacity#	(0.101, 0.230)	(0.144, 0.059)	(0.070, 0.063)	(0.134, 0.072)	(0.076, 0.132)

Table 10: Hausdorff distance between baseline skeletons and skeletons generated for the same model with different algorithms. The distance is given as a pair, where the first element is the distance from the baseline to the generated skeleton and the second element is the reverse. The symbol, #, denotes the usage of the sampling scheme introduced in section 7.2. Distances for the models **neptune**, **rotor**, and **wsm** can be found in appendix (table 12).

From this we can make some general observations.

Firstly, most of the distances is within the same general range without any extreme values. Since the distances are normalised, using the radius of the bounding sphere of the input model, we can interpret this to mean that the algorithm tends perform similarly for most inputs in terms of output quality, at least for those tested. We are unable to say much about the distance of one direction over the other, as it seems this changes from input to input.

Of the algorithms, **recomp** seems to generate skeletons that are nearest the baseline skeletons, which agrees with renders we have shown. This makes sense since this algorithm is the most similar to LSS. The **recomp-static** algorithm seems to compare consistently worse than **recomp**.

The expand-type algorithms tend to perform slightly worse than **recomp**. Of the three variations, **expand** seems to produce the better results, closely followed by **expand-capacity** and then **expand-continuous**, with **expand-capacity** and **expand** being very similar. This is not conclusive from this data alone, but intuition tells us that this should indeed be the case.

It might be the case that **expand-capacity** only really outperforms **expand-continuous** where **expand-continuous** fails, as can be seen on **human_hand** where we see a large difference in distance between the two. This is in agreement with what we found when we rendered the skeletons together in figure 17, where the skeleton of **expand-continuous** is very poor but **expand-capacity** still does well.

Lastly, we see that introducing sampling to **expand-continuous** had little affect on quality since the distances are largely unchanged.

	fertility	human_hand	temofoam	wsv	draco	neptune	rotor	wsm
LSS	171.8	26.6	59.3	137.5	174.2	100.3	5.3	31.5
recomp	13.5	5.4	313.0	42.5	37.9	19.8	4.6	7.7
*-static	8.7	4.6	1063.8	35.2	35.4	16.2	3.9	5.5
expand	4.3	2.2	109.1	14.5	11.3	5.7	2.0	3.3
*-continuous	4.1	2.1	52.3	14.0	7.6	4.5	1.7	3.1
*-capacity	4.2	2.1	60.6	13.9	7.9	4.9	1.8	3.2
*-capacity#	3.3	1.6	27.3	10.1	6.1	3.6	1.2	2.5

Table 11: Total runtime of each algorithm on a variety of models. The symbol, #, denotes the usage of the sampling scheme introduced in section 7.2. All measurements are given in seconds.

In table 11, we show the runtime of each algorithm on a number of models. we should immediately notice that our algorithms are generally orders of magnitude faster than LSS. An exception to this is **recomp** and **recomp-static** on voxel-grids where they sometimes run even slower than LSS, but it has already been explained why this happens. The **temofoam** model is also problematic for **expand** but to a much lesser degree. As previously discussed, the thin tube-like nature of **temofoam** means our choice of α is too high for this specific model, and it could be lowered to increase performance across the board.

As expected, we find that **recomp-static** runs faster than **recomp** but we only see a slight decrease in runtime. Interestingly, the runtime of **recomp-static** on **temofoam** is extremely high. We have found that something similar happens on **wsv** for high values of α, as can be seen on figure 33b in appendix, where a large jump in runtime happens for $\alpha = 256$. The reason as to why this only happens for **recomp-static** is unknown at the time of writing this.

The algorithm **expand** and its variations are generally the fastest, taking only a few seconds on some of the models. As expected, **expand** is the slowest, **expand-continuous** the fastest, and **expand-capacity** is somewhere in the middle.

Finally, the runtime of **expand-continuous** with sampling is very promising, giving low runtimes even for the most problematic model, **temofoam**.

In addition to the above measurements, we also perform qualitative evaluation of generated skeletons. Since the shapes we have examined so far have been complex, in the sense that it is not always clear how the skeleton should look, we have chosen to construct a simple cone shape and examine the quality of its skeletons.

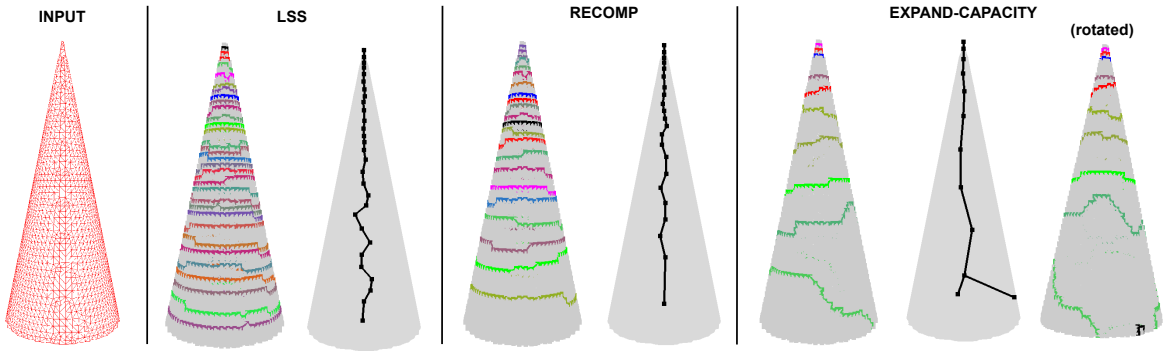


Figure 24: A simple cone shape, shown as wireframe, along with the packed separators and extracted skeletons for LSS, **recomp** and **expand-capacity**.

On figure 24 the results performing skeletonization on the cone shape is shown, in the form of both visualisation of packed separators and extracted skeleton using LSS, **recomp**, and **expand-capacity**. Of note is that the surface of the cone shape consists of evenly sized triangles, in order to avoid biasing any of the approaches. Let us then consider the performance of the algorithms. We see that LSS packs many tight separators, giving a mostly straight line through the centre of the cone, as expected. For **recomp** the skeleton also looks good, but we note that there are fewer separators along the shape, and that the skeleton ends higher up than for LSS. For **expand-capacity**, things are more complicated.

There are really two problems at work on **expand-capacity**, that leads to the erroneous skeleton. As can be seen on the rotated cone, a tiny separator has been found, that seemingly captures no actual feature of the shape. The cause of this lies at the high levels of the multi-scale graph.

As can be seen on figure 25 (A) the high levels of the multi-scale graph can be simplified to such a

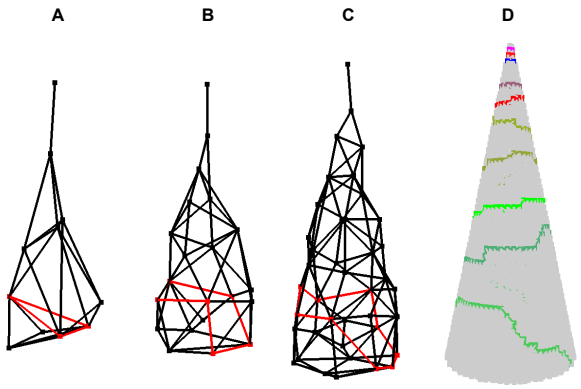


Figure 25: A visualisation of a "bad" separator (red) as found on a low resolution (A), expanded to higher resolutions (B-C) and as it appears on the final results (D)

degree that while the shape is retained, the graph is distorted enough that separators can be found in unexpected places. One workaround to this is to enforce greater balance when searching for separators, but this would not address the core issue.

Also seen on figure 25 is the process of finding a separator that, when expanded to the input, is far from the expected tight bands of LSS. The separator is indeed a tight band as found on (A), however not in the direction we would expect on the input. As the separator is expanded and minimized (B-C) it retains this jagged structure.

Although **recomp** also works on these high levels of the multi-scale graph, it does not seem to encounter these problems since it recomputes separators as LSS would on the actual input.

Once a "bad" separator has been found, it is very hard for the algorithm to discard them. In fact, it is only possible to do so when packing. But as we see on both **recomp** and **expand-capacity**, the density of separators is much lower than that of LSS. There appears to be large bands between separators, and also along the bottom of the cone, where no separators are found. If an area has a bad separator as the only separator, there is no competition when packing and the bad separators end up being part of the output.

The reason for these empty bands can be seen on figure 25 (B-C) along the bottom. Note that at low resolution, there might not exist separators that capture the lowest parts of the cone, but at higher resolutions the path around the cone may be too great for a restricted separator to be grown. This same mechanism applies to the areas between separators, but is most notable near the bottom.

We note that these extreme cases have not been observed on the remaining models. However, they do give an indication of scenarios that could happen, and so we will reflect on how to alleviate these issues in section 9.

We find that our **recomp** based algorithms generate nice skeletons, avoiding some of the issues that **expansion** based algorithms face in terms of separator quality. However, they are slower to compute, especially noticeable for voxel-grid inputs and for input such as **temofoam**. In general we find that **recomp-static** performs marginally better than **recomp**, but for **temofoam** the performance of **recomp-static** is much worse than any of our suggested algorithms.

The **expansion** based algorithms are much faster to compute, especially with sampling enabled, and in many cases produces skeletons of decent quality. It does however have issues that degrade output quality, which is especially noticeable on the cones.

We believe **expand-capacity** has the best trade off between quality and speed, and also the most potential for future improvements (see section 9). The fact that packing is fast also provides us with opportunities for further variations.

9 Future Work

As seen when evaluating the quality of skeletonization applied to a simple shape in section 8, expanded separators seem to be more jagged than those found using LSS. The heuristic used when shrinking aims to smooth out separators, but when a separator is grown at a coarse resolution, where it is possible that no smooth band exists, the separators also tend to become jagged after expansion. This can be explained by the fact that an expanded separator may not have much room for the heuristic to actually smooth it. One way to counteract this is to increase the size of the separator during expansion, by including also the neighbourhood of the separator. This concept is illustrated on figure 26 where a separator (C) is also expanded to include the neighbourhood of that separator on a lower level (orange vertices on (D)). When shrinking the separator, the heuristic then has more room for making the separator not only smooth but also more representative of the feature it should capture, as seen on (E).

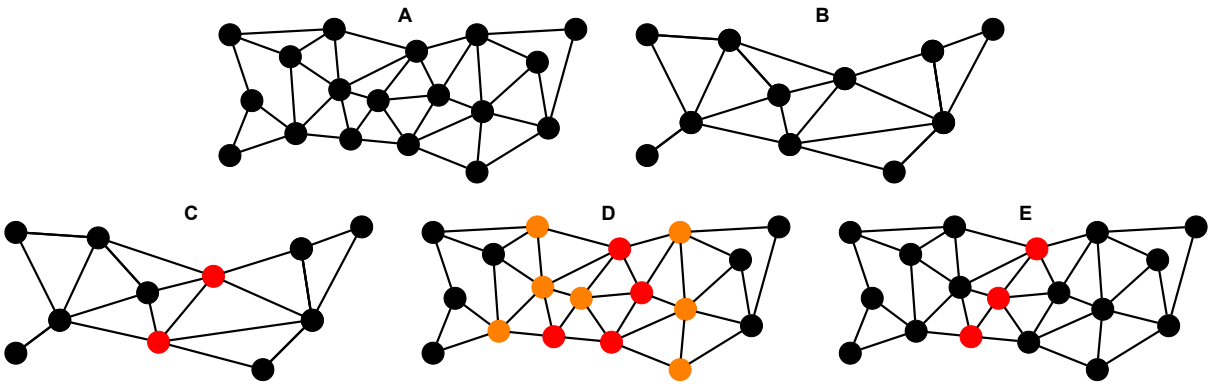


Figure 26: An input graph (A), its simplification (B), a separator found on the simplified input (C), the separator (red vertices) and additional vertices (orange vertices) after expansion (D), and the separator after shrinking (E).

In section 4.1.2, we showed an alternative way of naively shrinking a separator. For our variations where packing is not a computational bottleneck, it could be advantageous to apply several ways of shrinking a separator, effectively generating multiple minimal separators from a grown separator, and then let the packing be responsible for choosing the best. This idea can be combined with the idea of increased expansion, since there may not be overlap when choosing minimal separators in different ways on the increased expanded separators.

In addition, choosing minimal separators along the borders of additionally expanded separators may also allow for separators to be found in the areas "between" bands where no separators are otherwise found. To extend upon this idea, one could also consider "projecting" even further into this band, by also creating separators as those vertices adjacent to a minimal separator along the border but not in the expanded separator.

Since computing separators is expensive, even more so if we increase α, these approaches may serve to be fast ways to increase the separator count, so that packing can hopefully give a good result, without increasing computation cost significantly.

We have found that choosing a good value for α is not a trivial task, especially if one wants a value that is good in general. As we saw, a low value might save a lot of computation on thin models but is also

capable of missing features. An alternative approach to choosing a suitable value, perhaps depending on properties of the input or dynamically changing value during the run to account for some of these issues may be worth investigating.

As briefly mentioned in section 7.2, adapting **recomp** and its variation to allow for cross-level sampling is not entirely obvious. However, we have seen that sampling can make a very large difference computation time, and that the skeletons generated by **recomp** avoid some of the issues that **expand** and its variations encounter. Thus it seems that an adaptation of **recomp** to allow for cross-level sampling should give large benefits to running time, while achieving high output quality, at hopefully little change to the algorithm.

In section 7.1, we examined how changing the multi-scale graph, by altering how much we lower the resolution, changed the output and running time of the multi-scale approaches. Although changing the reduction factor outright does not seem beneficial, we have not investigated how different strategies for generating multi-scale graphs may affect the outcome. Things to consider include varying reductions depending on parameters of the input, or even between levels of the multi-scale graph, as well as methods that apply more operations than just edge contraction as described in [4]. In section 6.1.1 we also briefly considered the effects of generating the levels recursively, that is each level is a simplification of the previous, or by simply simplifying the input again at a higher degree of simplification. Since we observed that the non-recursive approached generated seemingly more representative multi-scale graphs, at the cost of additional computation and a not so clear relation between levels, there is also merit to considering if the recursive method could be altered to be more akin to the non-recursive method.

Intuitively, the recursive method cannot be guaranteed to perform the same simplifications, since we may end a round of contractions prematurely, having achieved the desired degree of simplification. Since the next level of simplifications do not take into account the remainder of the previous round of simplifications, it is possible to contract edges that the non-recursive method would otherwise consider locked. An idea is thus to keep track of the matching when generating the multi-scale graph recursively, so that it picks up from where it left off.

10 Conclusion

In this project we have described the skeletonization problem and the recent contribution using local separators presented by J.A. Bærentzen and E. Rotenberg. We have examine how the algorithm may be adapted for a multi-scale approach, in order to increase practical performance.

We have accounted for how we measure the quality of our output by using the Haussdorf distance to and from skeletons generated by the original algorithm, as well as the data used for testing performance.

We have presented two general approaches for adapting the algorithm to a multi-scale approach, one based on recomputation on the input, as well as one based on transforming from lower resolutions to higher. We have analysed and implemented both algorithms, and based on our observations, also presented variations of these, with the goal of improving certain aspects and performance on certain inputs.

Finally, we find that our suggested multi-scale approaches massively improve practical performance, for certain inputs even by orders of magnitude, at slight cost of output quality. We have also presented observations regarding parameter-tuning and sampling, as well as a wide array of results and measurements across different inputs, and our reflections regarding strengths and weaknesses, and how to possibly alleviate these in the future.

References

[1] Nicu D. Cornea, Deborah Silver, and Patrick Min. "Curve-Skeleton Properties, Applications, and Algorithms". In: *IEEE Trans. Vis. Comput. Graph.* 13.3 (2007), pp. 530–548. DOI: 10.1109/TVCG.2007.1002. URL: https://doi.org/10.1109/TVCG.2007.1002.

[2] Andreas Bærentzen and Eva Rotenberg. *Skeletonization via Local Separators*. 2020. arXiv: 2007.03483 [cs.CG].

[3] Achi Brandt. "Multiscale Scientific Computation - Review 2000". In: (Jan. 2000). DOI: 10.1007/978-3-642-56205-1_1.

[4] Mario Botsch et al. "Geometric Modeling Based on Polygonal Meshes". In: *Eurographics 2008 - Tutorials*. Ed. by Maria Roussou and Jason Leigh. The Eurographics Association, 2008. DOI: 10.2312/egt.20081055.

[5] Jacob Holm, Kristian de Lichtenberg, and Mikkel Thorup. "Poly-Logarithmic Deterministic Fully-Dynamic Algorithms for Connectivity, Minimum Spanning Tree, 2-Edge, and Biconnectivity". In: *J. ACM* 48.4 (June 2001), pp. 723–760. ISSN: 0004-5411. DOI: 10.1145/502090.502095. URL: https://doi.org/10.1145/502090.502095.

[6] *GEL library*. 2022. URL: https://github.com/janba/GEL (visited on 05/02/2022).

[7] Leif Kobbelt et al. "Interactive Multi-Resolution Modeling on Arbitrary Meshes". In: *Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '98. New York, NY, USA: Association for Computing Machinery, 1998, pp. 105–114. ISBN: 0897919998. DOI: 10.1145/280814.280831. URL: https://doi.org/10.1145/280814.280831.

A Appendix

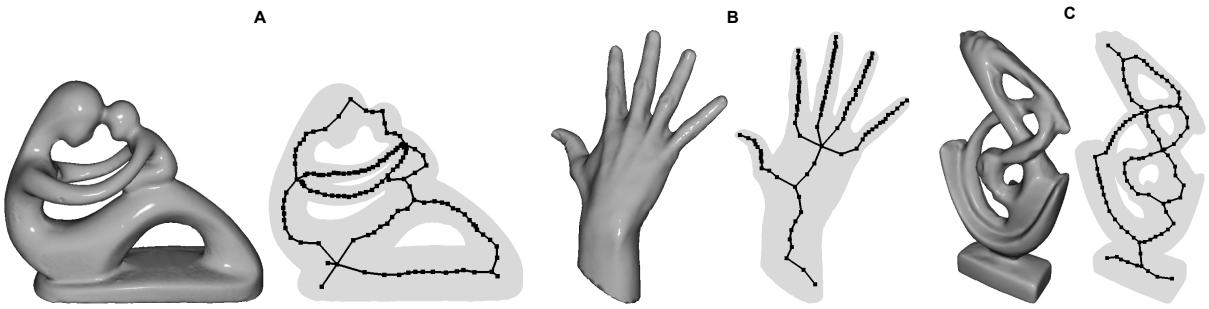


Figure 27: Skeleton output of running LSS on two meshes (a and b) and one voxel-grid (c). To the left of each skeleton is shown a shaded render of the model used to as input for the skelenozisation. The models are **fertility** (A), **human_hand** (B), and **wsv** (C)

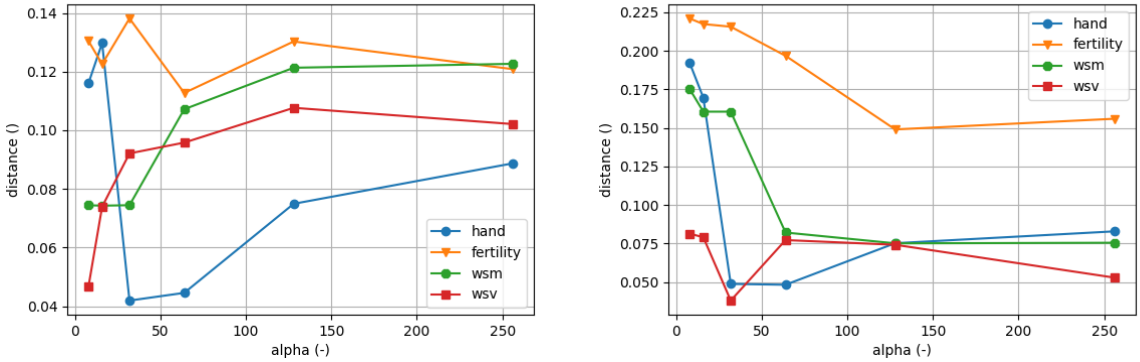


Figure 28: Hausdorff distance as a function of α for recomp. To the left, the distance from baseline skeletons to a skeleton of the same model generated with the algorithm. The right is the reverse.

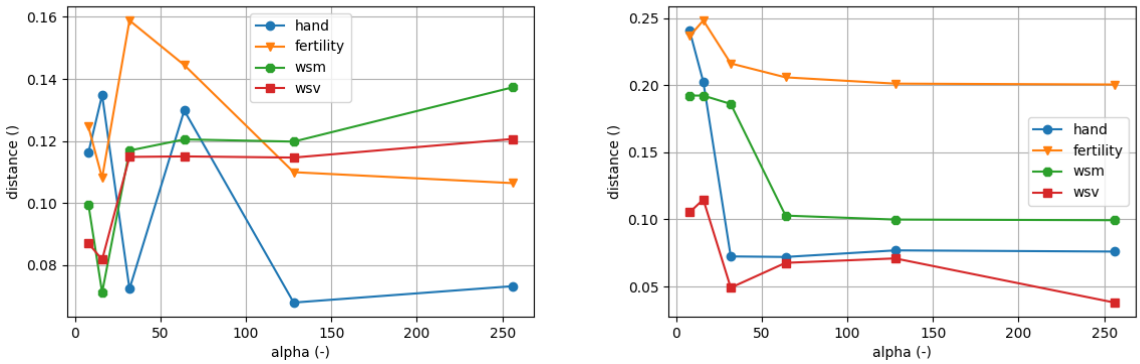


Figure 29: Hausdorff distance as a function of α for recomp-static. To the left, the distance from baseline skeletons to a skeleton of the same model generated with the algorithm. The right is the reverse.

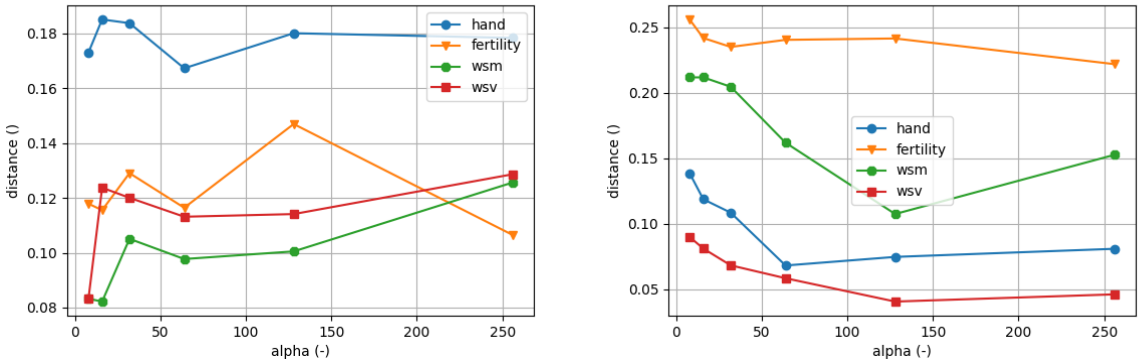


Figure 30: Hausdorff distance as a function of α for expand. To the left, the distance from baseline skeletons to a skeleton of the same model generated with the algorithm. The right is the reverse.

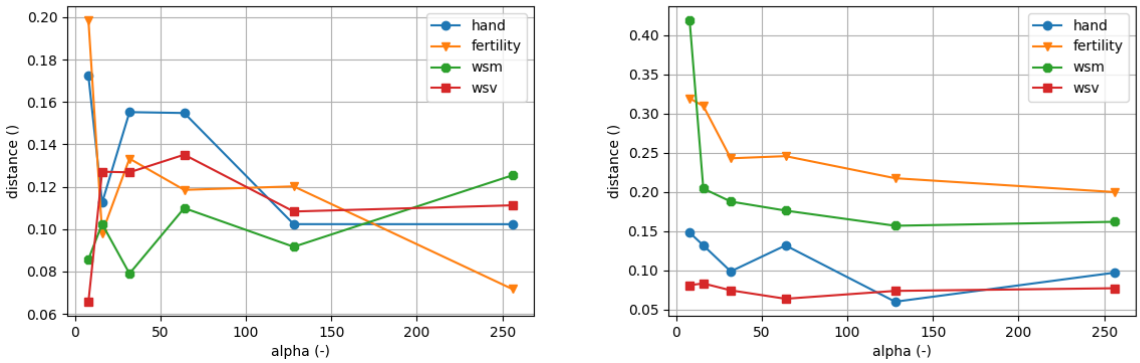


Figure 31: Hausdorff distance as a function of α for expand-continuous. To the left, the distance from baseline skeletons to a skeleton of the same model generated with the algorithm. The right is the reverse.

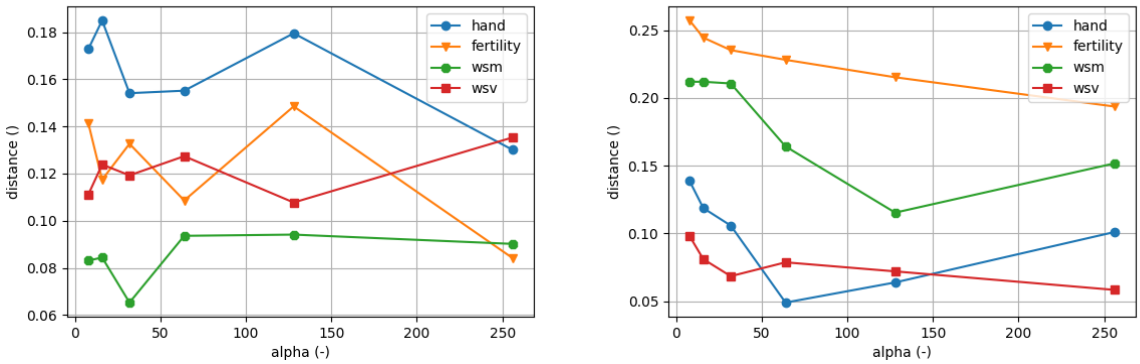
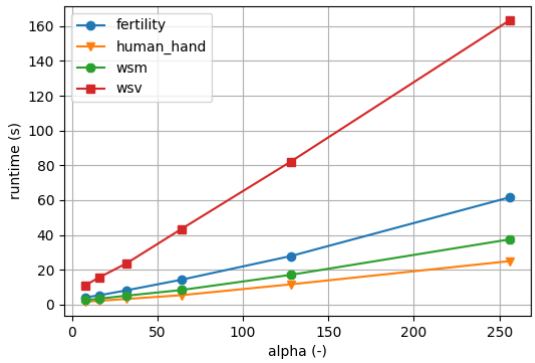


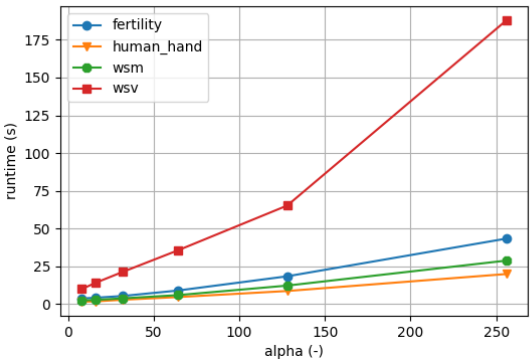
Figure 32: Hausdorff distance as a function of α for expand-capacity. To the left, the distance from baseline skeletons to a skeleton of the same model generated with the algorithm. The right is the reverse.

	neptune	rotor	wsm
LSS	(0.000, 0.000)	(0.000, 0.000)	(0.000, 0.000)
recomp	(0.074, 0.119)	(0.054, 0.051)	(0.098, 0.078)
*-static	(0.062, 0.187)	(0.065, 0.068)	(0.111, 0.108)
expand	(0.083, 0.097)	(0.056, 0.054)	(0.094, 0.161)
*-continuous	(0.082, 0.102)	(0.062, 0.051)	(0.099, 0.175)
*-capacity	(0.085, 0.101)	(0.061, 0.067)	(0.089, 0.163)
*-capacity#	(0.084, 0.104)	(0.059, 0.067)	(0.090, 0.188)

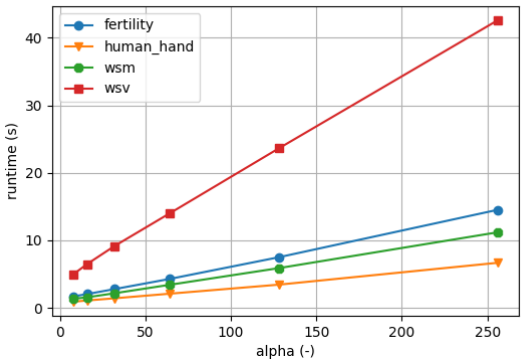
Table 12: Hausdorff distance between baseline skeletons and skeletons generated for the same model with different algorithms. The distance is given as a pair, where the first element is the distance from the baseline to the generated skeleton and the second element is the reverse. The symbol, #, denotes the usage of the sampling scheme introduced in section 7.2.



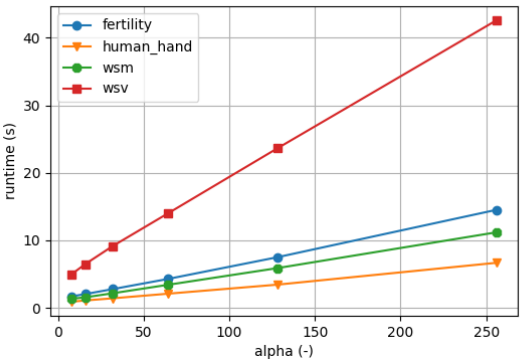
(a) recomp



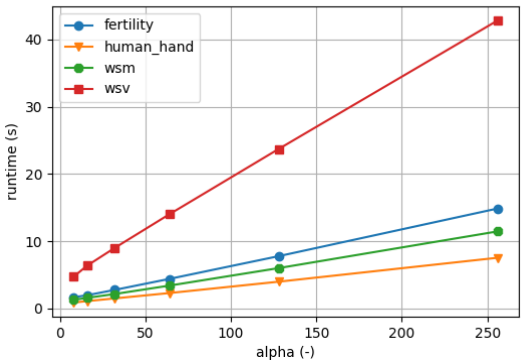
(b) recomp-static



(c) expand



(d) expand-continuous



(e) expand-capacity

Figure 33: Runtime as a function of α for each algorithm and their variations.

B Efficient Methods for Graph-Based Generation of Skeletons from 3D Data

Efficient Methods for Graph-Based Generation of Skeletons from 3D Data

Emil T. Gæde
Rasmus E. Christensen

Kongens Lyngby 2022

Contents

1 Introduction **2**

2 Skeletonization with Local Separators **2**

3 Method **3**

4 Preliminary Analysis **3**

5 Theory **4**
5.1 Euler Tour Trees 4
5.2 Extending ETT for General Graphs 7
5.3 The Dynamic Connectivity Data Structure 8

6 Implementation **9**

7 Integration **10**

8 Preliminary Results **11**

9 Optimisation **12**
9.1 Thresholding 12
9.2 Recency 13

10 Results & Discussion **14**
10.1 Further Work 16

11 Conclusion **17**

References **17**

Division of labour

This project has been made in close collaboration by both participants and both have been involved with developing the code and writing the report. The table below shows, which section each member has been primarily responsible for.

Primary Responsible	Sections
Emil (s164414)	1,4,5,6,7,9.2,10.1
Rasmus (s164408)	2,3,8,9.1,10

1 Introduction

The act of skeletonization is the process of generating a curve skeleton of a 3D shape, reducing it to a 1D representation. Curve skeletons are thus simplified representations that maintain structural information about the original shape.

Curve skeletons are elusive in the sense that; what information they should retain, what features they should capture, and what makes a "good" skeleton is highly dependant on the application. Some of these applications include computer graphics, medical image analysis and more [1].

In [2] A. Bærentzen and E. Rotenberg propose a new algorithm for computation of curve skeletons based on the concept of local seperators. The algorithm is seemingly able to capture fine details of the shapes, while making relatively little assumptions about the input, requiring it simply to be a spatially embedded graph.

Although the resulting skeletons are good, the algorithm is not currently implemented with practical performance in mind.

The goal of this project is thus to alter and implement the algorithm in such a way that the overall algorithm and output remains unchanged, but the practical performance is improved.

We will analyse areas of improvement, propose and implement alterations of data structures and subroutines, reflect on the results of these improvements, and comment on options for further optimisation.

2 Skeletonization with Local Separators

Skeletonization with local separators[2], or LS, is an algorithm for computing curve skeletons of shapes. The algorithm works on shapes that can be represented as a spatially embedded graph. The algorithm processes the graph in multiple steps, which will briefly be described in this section.

The main idea of the algorithm is to find a non-overlapping set of *local separators*, from which a skeleton can then be extracted.

The first step of finding the non-overlapping set of local separators is to sample vertices of the input graph. From each of these vertices, we will then find a local minimal separator. This creates a set of overlapping separators. Using a set-packing approximation algorithm with a greedy heuristic, the set is converted to a set of non-overlapping separators. All vertices not part of a separator are then assigned to the closest one. With all vertices part of a separator, a skeleton is extracted as the quotient graph with separators defining the partitioning. The last step is to perform a "cleanup" of the skeleton by removal of *cliques* in the quotient graph.

For reasons explained later, we will primarily deal with the *finding of separators* phase of the algorithm, hence we will explain this step in more detail. Finding a local minimal separator consist of two primary steps; growing and shrinking.

To grow a separator, we begin in an initial vertex. We assign the vertex to the separator Σ and all of its neighbours to the front F, the front being vertices not in Σ but neighbours to vertices in Σ. From here we grow the separator by finding the vertex in F that is the closest to a bounding sphere that we maintain around Σ. The vertex is added to Σ, and the front and bounding sphere is updated. The algorithm ends when the front is disconnected, which is checked by running a BFS after each vertex has been added. Finally, the size of the components in F is used to rate the quality or possibly even reject the separator.

The next step is to shrink the separator so it becomes a minimal separator. This is done by iteratively removing vertices from the separator until is not possible to remove more. The final separator is dependent on the order in which vertices are removed. To get "smooth" separators, a heuristic that accounts for the geometry and bounding sphere of the separator is used to order the vertices.

3 Method

We will be running different experiments to aid in our analysis of the original and our new algorithms. All experiments are run on an AMD Ryzen 7 4800U CPU using all 8 cores. The original algorithm and new additions are all written in C++17 and compiled with gcc 11.1.0 using the highest level of optimisations (`-O3`).

The LS algorithm uses a pseudorandom generator to sample nodes from each of which a separator will be grown. Because of the parallelism of the growing step, the sampled set of nodes is not deterministic. To compare two different approaches it can be useful to test the time it takes to find separators when growing from the same set of nodes. We will be using a method where we run the algorithm with a *presampled* set of nodes to achieve this. Because our algorithms create the same separators as the original algorithm for any given set of starting nodes, the sampling of the two methods should be identical. Thus, to generate the set we simply run the original algorithm on a model and record each node where a separator was grown. This will then give us a sample of starting nodes that could "naturally" occur for the given model. It should be assumed that whenever we compare the runtime of two algorithms for a given model, we have used a set of presampled starting nodes.

The LS algorithm works on multiple graph representations of models. We will mainly be testing the algorithm on models used in the original paper [2] provided by the authors. These models primarily include meshes and a few voxel-grids. We will also use a single model of a botanical tree generated from a point cloud. We have two groups of models.

The first group contains distinct models with different features and number of vertices. For instance, **temofoam** is a model consisting of many thin tubes crossing over each other while bunny is an almost spherical model with ears. This should test the algorithms in different scenarios that are more or less difficult.

The **wsm** and **wsv** model-sets are meshes and voxel-grids respectively, both of which are generated from the **wood_statue** model. Each set includes models of increasing amount of detail and vertices and will be used to test how the algorithm scales with the number of vertices.

4 Preliminary Analysis

An issue with LS is that process of finding and packing local separators is very slow for large separators [2, pg.22]. To improve the performance of the algorithm, we begin by performing an experimental evaluation in order to determine which part of the algorithm is the primary bottleneck.

To do this, we will divide the LS algorithm into finer parts. Findings by Bærentzen and Rotenberg indicate that the running time is largely dominated by the computation of separators [2, pg.20]; thus, we will focus on this aspect. For growing a separator we measure the time spent on finding the next vertex to add, as well as the time spent on checking connectivity. For shrinking a separator, we measure the time it takes to calculate the heuristic and order the vertices accordingly, and the time spent removing vertices from the separator. Results of our measurements can be seen on table 1 for a variety of models.

Model	Computing	Packing	Find Closest	Connectivity	Ordering	Removing
neptune	120.141	51.016	39.081	655.995	182.336	8.689
rotor	5.829	3.126	2.767	33.457	3.845	0.584
wsv90	659.548	10.776	98.661	4287.46	658.108	15.263
wood_statue	61.956	8.862	22.361	357.338	74.152	4.229

Table 1: Measurements of time spent on sub-processes on different models. Computing and Packing measured in wall time remaining measurements in CPU time.

The measurements show that indeed computing the separators takes a large part of the time. Specifically, checking the connectivity after adding a vertex to the separator seems to be the largest bottleneck.

Thus, for the remainder of the project we will focus on how to reduce the time spent on connectivity checking by modifying the algorithm to no longer use a BFS each iteration, but rather employ a dynamic connectivity data structure.

5 Theory

In this section we will give a description of the data structures used in the project to solve the dynamic connectivity problem.

The dynamic connectivity problem that we concern ourselves with can be said to be that of fully dynamic connectivity. We wish to construct a data structure that lets us insert and remove edges (and by extension vertices) from a graph, such that we can quickly answer queries related to the connectivity of the graph, ie. are two vertices in the same component.

The main objective of this section is a give description of a data structure for fully dynamic connectivity in general graphs, originally presented in [3]. To give a better understanding of the workings of the data structure, we will first describe methods for dynamic connectivity in forests, and representation of trees.

The specific data structure has been chosen for its relative simplicity, as well as previous experimental performance [4].

Throughout the section, we will let n denote the number of vertices in the graph.

5.1 Euler Tour Trees

To solve the issue of fully dynamic connectivity in forests we can use Euler tour trees [5].

The idea is to represent an arbitrary undirected tree by the Euler tour of the tree rooted at an arbitrary node. Such a tour exists if we replace each undirected edge u, v with two directed edges: $(u, v), (v, u)$. We then store the Euler tour of the tree in a balanced binary tree structure where the vertices of this binary tree represent the edges of the Euler tour in the original tree. The edges are stored in visitation order, such that an inorder traversal of the resulting tree gives the Euler tour.

In addition, we store the first visit to each vertex, v, in the Euler tour as (v, v) in the balanced tree - the purpose of this will be described later. We note that variations of this representation exist, in which we store not the edges of the Euler tour, but rather the "visits" of vertices (as originally proposed). However, our representation using edges allow us elegant operations that will be described later. Note that the resulting Euler tour tree representation of a tree of n vertices contains $n + 2(n - 1) = O(n)$ vertices.

A visualisation of a tree and a possible Euler tour representation can be seen on figure 1.

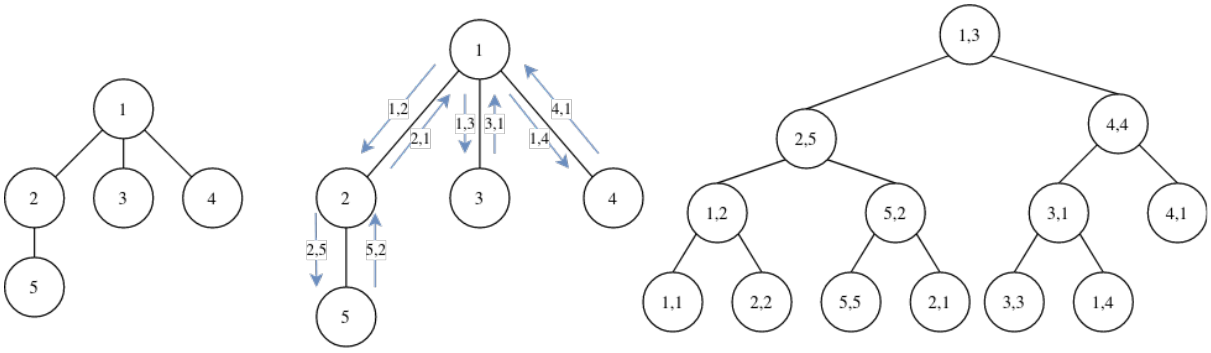


Figure 1: A tree, an Euler tour of the tree and the Euler tour representation

Lastly, we store a BBST, using edges as keys that store pointers into the balanced tree representation to provide fast access for operations.

Before delving into the operations, let us briefly consider ourselves with the intuition behind the representation. As mentioned before, an inorder walk of the Euler tour tree gives the Euler tour, and in general, we can think of the Euler tour tree as a way to represent a sequence. Since the sequence is determined by the inorder traversal of the Euler tour tree, there is a tight relationship between the structure of the Euler tour tree and the sequence it represents.

The commonly known `split` and `join` operations on balanced binary trees allow us to manipulate this structure, while maintaining the internal order of the Euler tour tree. Thus, we can essentially think of these operations as working on the sequence rather than the tree.

Consider, as an example, if we would like to re-root the Euler tour such that the Euler tour tree represents the same tour starting from a different vertex. Since the Euler tour is cyclic, this re-rooting is simply a cyclic shift of the sequence the Euler tour tree represents. If we retain this idea of manipulating the sequence, it is then easy to see that a cyclic shift can be achieved by a `split` at the desired starting point, and then a `join` to recombine the subsequences in the desired order.

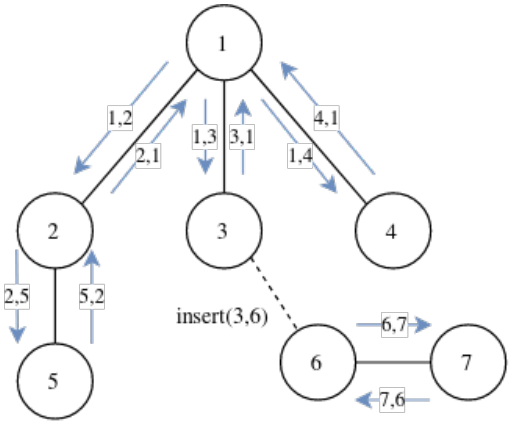
With this in mind, we will describe how to utilise `split` and `join` to perform `insert` and `delete` operations on Euler tour trees.

Let us consider insertion of an edge (u, v), where $s(u)$ denotes the sequence of the Euler tour containing u cyclically shifted as previously described so that (u, u) appears first in the sequence and conversely for $s(v)$.

Since $s(u)$ ends in an edge that goes to u we can extend this tour with the edge (u, v). Likewise we can extend $s(v)$ with the edge (v, u) and lastly join the two sequences so that the result is the sequence $s(u), (u, v), s(v), (v, u)$. We note that the number of splits and joins to re-root the two tours and join them in this fashion is constant.

Visualisation of how insertion relates to the joining of sequences can be seen on figure 2.

When deleting an edge in a tree, we know that this must disconnect the tree into two. Consider deletion of edge (u, v) that appears in the middle of some Euler tour. Prior to (u, v) occurring in the sequence there is some traversal of the tree ending in u. A subsequence then begins traversing the tree until reaching (v, u) before continuing. This subsequence between (u, v) and (v, u) is the Euler tour of one of the components after deletion. If we concatenate the remaining sequences, we obtain the other

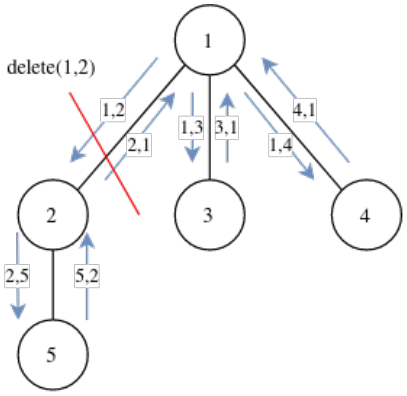


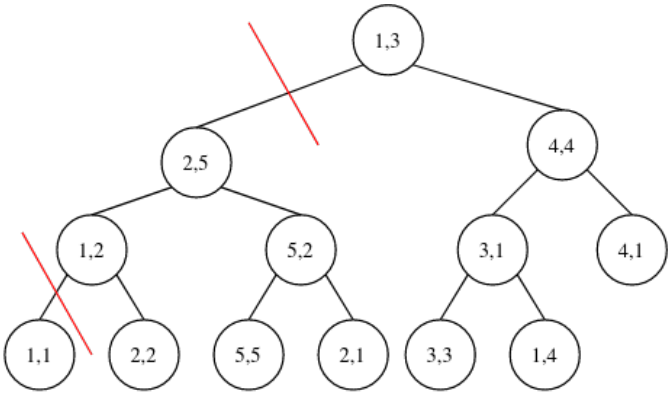
(3,3)(3,1)(1,4)(4,4)(4,1)(1,1)(1,2)(2,2)(2,5)(5,5)(5,2)(2,1)(1,3)(3,6)(6,6)(6,7)(7,7)(7,6)(6,3)

Figure 2: Visualisation of the insertion of an edge (3,6) in the Euler tour representation

component. We note that (v, u) might occur prior to (u, v) in the sequence, but we can simply delete (v, u) instead and proceed as prior.

Visualisation of how deletion relates to extracting subsequences can be seen on figure 3.





(1,1)(1,2)(2,2)(2,5)(5,5)(5,2)(2,1)(1,3)(3,3)(3,1)(1,4)(4,4)(4,1)

Figure 3: Visualisation of the deletion of an edge (1,2) in the Euler tour representation

What remains is then only how to perform connectivity queries. Intuitively, if we can obtain the sequence a vertex belongs to, we can find the first element of the sequence and compare these.

The auxiliary BBST allows us to perform the lookup, and we know that the first element of the sequence that the Euler tour tree represents exists as the leftmost child of the root. However, to find the leftmost child of the root we have to traverse to the root initially, and this is just as fitting a representative for comparison. Thus we can simply look up both vertices, traverse the Euler tour trees to the root and compare these. To vertices are then connected if they share the same root.

Since we are storing the Euler tour tree in a balanced binary tree structure, the height is $O(\log n)$, as the number of vertices in the Euler tour representation of a tree of size n is $O(n)$ as previously shown.

We know that `split` and `join` can be performed in $O(\log n)$ time for balanced binary trees and that re-rooting, inserting, and deleting uses only a constant number of these operations. In addition, there may

be lookups related to finding the relevant vertices of the Euler tour trees but since these are performed in a BBST the total time of any one of the described operations is $O(\log n)$.

Thus, we can use Euler tour representations of trees to solve the fully dynamic connectivity problem in forests using $O(\log n)$ time for updates and queries. We will then examine how these ideas can be extended to solve the problem for general graphs.

To fully understand the dynamic connectivity structure of [3] we will first show how to extend the structure for trees into one for general graphs, and then how to further enhance it to achieve better theoretical bounds.

5.2 Extending ETT for General Graphs

Since insertion in trees always connect components, and deletes always disconnects them it is sufficient to represent a forest of Euler tour trees. In general graphs, however, we might insert an edge between two vertices that are already in a connected component and we may delete edges that do not disconnect anything.

The idea is then to maintain a spanning tree of each of the components, and augment the Euler tour trees with information about the edges of the component that are not in the spanning tree, s.t. we can maintain the forest under the updating operations.

Let us first consider how to extend the `insert` operation. Our Euler tour trees allow us to query if the vertices are already connected. If this is the case, we then consider the new edge a "non-tree" edge. For sake of the later described `delete`-extension, we will store in the nodes of the Euler tour tree what non-tree edges are adjacent to the vertex of the graph that the node represents.

Since insertion of a non-tree edge does not change connectivity, updating this information is all we have to do. Likewise, since insertion of tree-edges do not change what non-tree edges are adjacent we simply insert as prior.

We note that maintaining and updating this information does not change the complexity of insertions, leaving it at $O(\log n)$ complexity.

Let us then consider the `delete` operation. When removing an edge that is part of a spanning tree, it could be the case that there is some non-tree edge in the component that could re-connect the tree. Thus, we must search through the non-tree edges of one of the resulting Euler tour trees and see if any of the non-tree edges connects to the other Euler tour tree. Let us briefly expand upon how to do this in a practical manner.

Since Euler tour trees use balanced binary tree structures, we can augment these binary trees with information about subtrees that we can maintain when updating the tree. In our case, we will maintain information about the sizes of subtrees, a flag indicating if there are adjacent non-tree edges to the subtree, as well as adjacency-lists for adjacent non-tree edges. On figure 4 a small example showing a component, the spanning tree and non-tree edges along with how the Euler tour tree is augmented to reflect these.

To examine if a non-tree edge reconnects the trees, we simply check the root of the tree that the other endpoint of the edge resides in. Note that since the deleted edge previously was in the connected component, it must either connect to the same Euler tour tree that we are searching through or the one we wish to reconnect to.

When deleting an edge in a Euler tour tree we get two resulting trees. Since we maintain information about their sizes, we can choose to only search through the smaller tree. The flag denoting if there

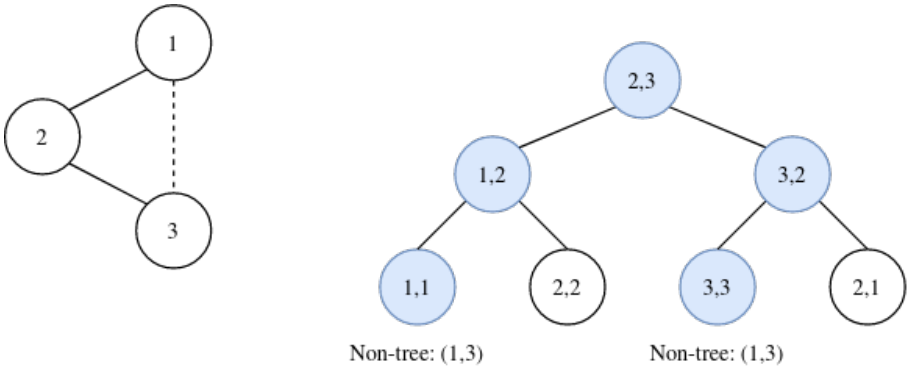


Figure 4: Left: small component with spanning tree shown in solid lines. Right: Euler tour tree of the component with augmentations to reflect additional information. Blue nodes have non-tree edges in their subtree.

are adjacent non-tree edges in the subtree then allow us to only search through the subtrees that may contain reconnecting edges, providing additional performance in practice.

We note that it takes $O(\log n)$ time to find and determine if a single non-tree edge reconnects - but we may have to check every non-tree edge if the deletion truly disconnected the component.

Since removal of a non-tree edge does not change connectivity, we simply update the information in the Euler tour trees nodes that we updated on insertion.

Since the Euler tour trees are spanning trees of the components of the graph, our previously described connectivity query needs no modification.

5.3 The Dynamic Connectivity Data Structure

We will then introduce the dynamic connectivity data structure presented by Holm, de Lichtenberg and Thorup[3]. The data structure builds upon the concepts of Euler tour trees for general graphs, but aims to improve upon how we search for replacement edges during deletion.

The idea is to maintain a spanning forest of the graph, along with a hierarchy of subforests that allow for systematic searching. Formally, we associate each edge with a level and maintain a hierarchy of spanning forests s.t. the forest F_i is the forest induced by edges of level at least i and $F_0 \supseteq F_1 \supseteq \cdots \supseteq F_{max}$. We will later argue that the maximum level is $O(\log n)$. We note that non-tree edges are only present on the level they are associated with.

Let us then consider how `insert` and `delete` are changed. When inserting an edge, we insert it at level 0. When deleting a tree edge of level i, we delete it from every forest of level $\leq i$, and then we search for a replacement on level i. When doing so, we push every edge of the smaller spanning tree it connected to level $i + 1$. When searching through the non-tree edges, we also push every non-replacement edge to the next level. If no replacement is found on level i, we try on level $i - 1$, and repeat until either a replacement is found or we have failed to find one on level 0. Since level 0 is the entire spanning tree, a failure to find means no replacement edge exists and we can report that the component was disconnected.

On figure 5 the hierarchy of forests is visualised during a series of deletions on a line graph.

Intuitively, if we have searched a non-tree edge and found it not to reconnect, we push it to the next level. Thus, this edge cannot reconnect a deletion unless the deletion occurs at least at the level of the

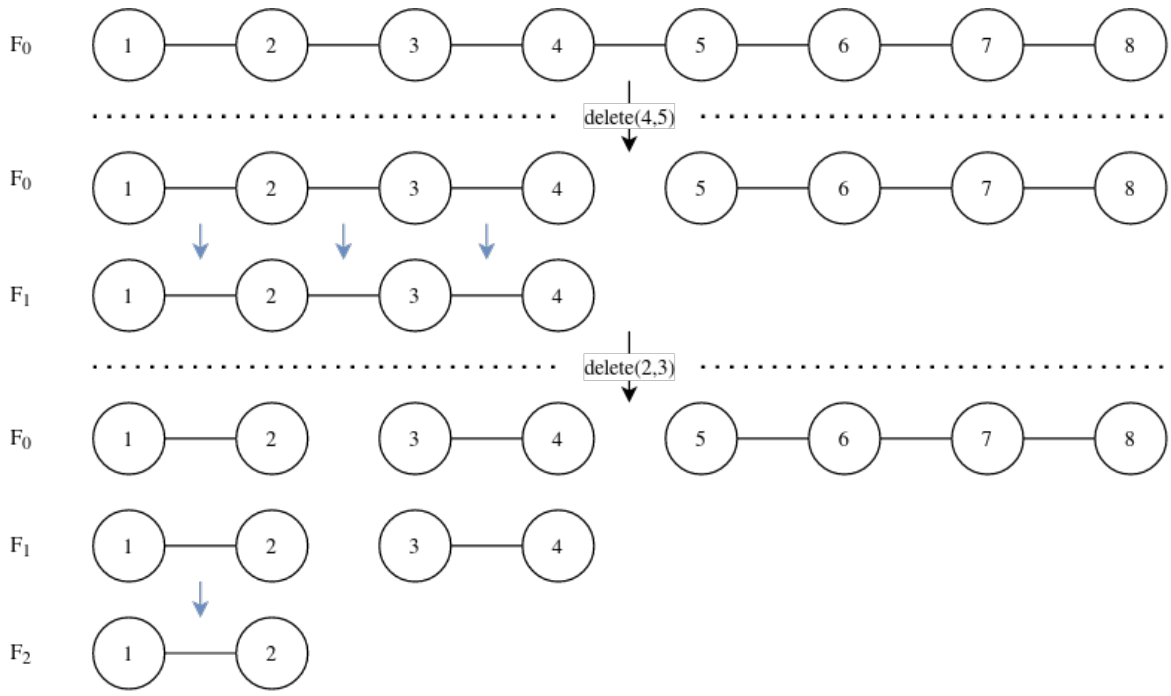


Figure 5: The spanning forests of the data structure on a line graph through deletion of edges. Blue arrows indicate edges being pushed to the next level.

edge. This means for deletions on lower levels, we do not have to examine it once again.

Let us briefly examine how this pushing gives a $O(\log n)$ bound on the number of levels. Note that we only push when deleting, and only the edges of the smaller component. Thus, trees in F_i contain at most $\frac{n}{2^i}$ vertices, giving a maximum level of $O(\log n)$.

Since F_0 is the entire spanning forest, we can simply perform connectivity queries on this forest to answer for the entire structure.

It can be shown that the update operations then take amortized $O(\log^2 n)$ time[3], but let us briefly consider the intuition behind this result.

Inserting an edge at level 0 takes $O(\log n)$ time, but it may be pushed a total of $O(\log n)$ times each at cost $O(\log n)$.

When deleting an edge, we delete it from $O(\log n)$ levels each taking $O(\log n)$ time. We must then search for a replacement on $O(\log n)$ levels each taking $O(\log n)$ time plus the amortized cost of pushing edges.

6 Implementation

In this section we will specify some of the relevant details relating to our implementation of the dynamic connectivity data structure.

When implementing Euler tour trees, we use splay trees as the underlying balanced binary tree. Since nodes of the splay tree represent edges (and vertices) of the graph, we store both a `start` and `end` identifier, where for nodes representing vertices `start=end`. We also store the size of the subtree rooted at the node, as well as two flags. The first flag indicates if the subtree has adjacent non-tree edges, while the second flag indicates that the node is to be processed when searching.

For nodes representing vertices, this flag indicates that this vertex has adjacent non-tree edges.

For nodes representing edges, the flag indicates if this is the highest level at which the edge exists. If this is not the highest level at which it exists, we need not bother pushing it when searching, which helps us not perform unnecessary operations related to linking trees on levels which already contain the edge.

We store adjacent non-tree edges in linked lists while maintaining pointers s.t. we can perform deletion from the list in $O(1)$ time.

7 Integration

To make use of the dynamic connectivity data structure, we must change how each iteration of the LS algorithm determines the front.

Originaly, we simply added vertices to a set representing the front and then performed a BFS to determine the connected components. The dynamic connectivity structure, however, requires us to consider the edges of the front in each iteration.

When a vertex, v, is added to the separator we wish to add all neighbours of v to the front. The edges to be added are then those between any neighbour of v and a vertex in the front, excluding the edges between v and its neighbours that were previously not in the front since these would be removed in the next step. After inserting these edges, we must remove the edges that connect v to the front.

Thus, when adding a vertex to the separator we perform a number of insertions linear in the sum of degrees of the neighbourhood of v and then remove a number of edges linear in the degree of v.

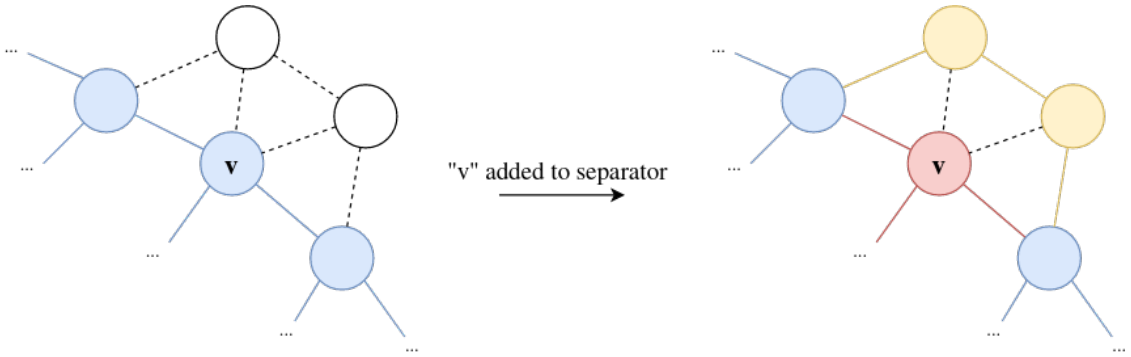


Figure 6: Visualisation of adding a vertex to a separator. Blue vertices and edges denote the front, red edges and vertices are those removed from the front, and yellow edges and vertices are added to the front.

On figure 6 a visualisation of what edges are affected by these insertions and deletions can be seen.

For sake of the algorithm we must also know the number of components and their sizes. To track this, we maintain a set of representatives where each representative is the root of an Euler tour tree in the current iteration. To determine these we add the representatives of each neighbouring vertex of v to a set in each iteration. Since the Euler tour trees are stored using Splay trees, these representatives may change with each update of the data structure, meaning we have to "update" previously found representatives by once again traversing to their roots.

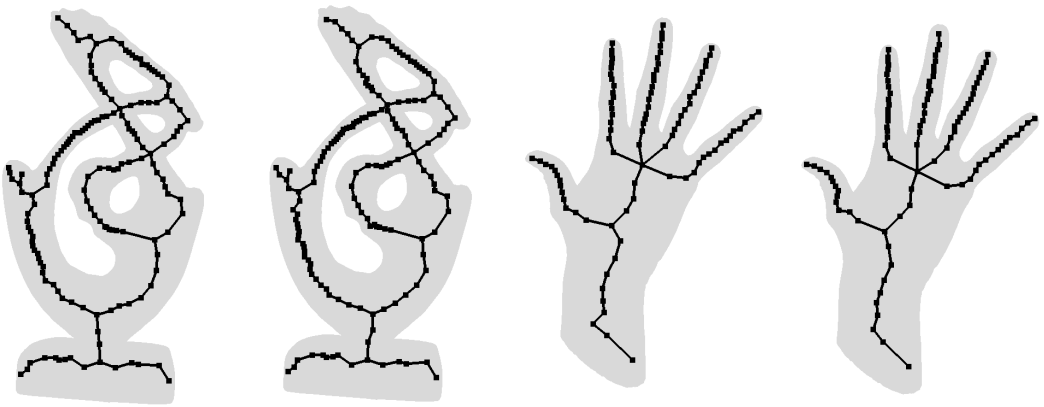
Since the roots of the Splay trees also maintain information about the sizes of the trees we also know the sizes of the components, allowing us to compute the front size ratio necessary for the algorithm to

determine if a separator has been fully grown.

To briefly argue that these changes to the algorithm do not worsen the theoretical bounds, let us consider the amount of work performed in each iteration. In the original algorithm we spent time linear in the number of edges and vertices in the front, while we now spend $O(\log^2 n)$ amortized time per edge inserted and deleted. We insert a number of edges bounded by the sum of degrees of the neighbourhood of v, denoted s, and delete a number of edges linear in the degree of v. The time spent on each iteration is then $O(s \log^2 n)$ which for $s = O(\frac{n}{\log^2 n})$ is no worse than the original algorithm.

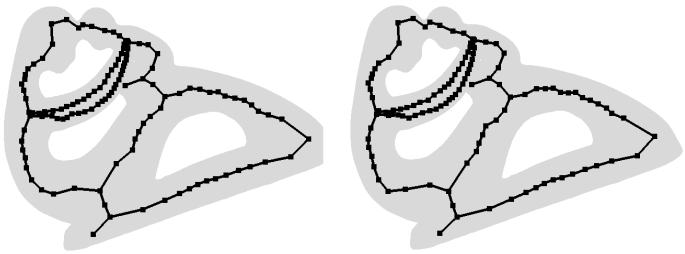
8 Preliminary Results

The change made to the LS algorithm is only that of checking connectivity in separators. Because of this, the produced skeletons should be the same as for both the original and the new algorithm. By looking at skeletons produced by the algorithms, as those show in figure 7, we see that the results are largely the same, with only minuscule differences due to sampling of starting nodes.



(a) wood_statue

(b) human_hand



(c) fertility

Figure 7: Skeleton comparisons of the original algorithm(left) and the skeletons generated with the use of the dynamic connectivity data structure(right).

We have also tested the new algorithm by running the original algorithm simultaneously and comparing results of all connectivity querries. This produces no errors so we conclude that the new algorithm works correctly.

9 Optimisation

The applied dynamic connectivity data structures is designed to work for a general case and is also not optimised for practical performance. For this reason, we suspect that is is possible to improve the data structure with two easily implemented optimisations. The changes to the data structure will not affect the results of queries; thus, the generated skeletons are unchanged.

In this section we will test and cover the two possible optimisations. Based on our findings, we will create an optimised version of of the algorithm with improved performance.

9.1 Thresholding

Keeping track of many forests and pushing edges in the dynamic connectivity data structure can require a substantial amount of work. The payoff is that once an edge is pushed a level, we potentially have to do less work in the future when looking for a replacement edge. In practice, this might not be a worthwhile investment once the size of trees reaches a certain limit. Intuitively, it should be fast to search through the tree when it is small, so we might improve performance if we limit the minimal size of trees such that edges are never pushed past a certain point.

To determine the minimal size of trees we use a threshold and perform an empirical experiment to determine its optimal value. We expect that the value is hardware dependent since cache sizes and the processor could affect results. The optimal value might also be different for different kinds of data since the pattern of insertions and deletions greatly affect how often edges are being pushed. For this reason we will perform the experiment for both a voxel-model and a mesh-model.

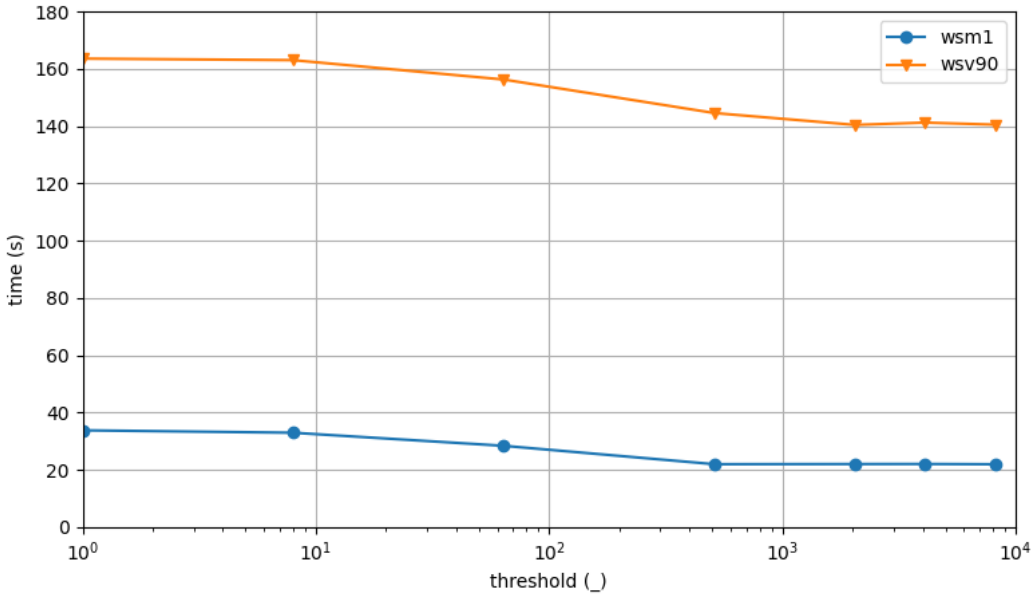


Figure 8: Time to find separators on a mesh graph and on a voxel graph of *Wood Sculpture* as a function of the threshold value.

Figure 8 shows the time to find all separators on the wood sculpture as both a voxel-grid and a mesh. Here we clearly see that the runtime decreases as the threshold value increases until a certain point where the runtime stays constant, i.e., the runtime approaches a certain value as the threshold goes

towards infinity. What this effectively means is that it is faster to never push edges in the dynamic connectivity data structure.

If we never perform any push operations, it makes no sense to have multiple forests in the data structure. What we are effectively left with is a dynamic connectivity data structure, which contains a single forest that can answer if two nodes are connected in $O(\log n)$ time. When performing a delete operation, we will now always have to look for a replacement edge in that single large forest. However, Because we store information of non-tree edges of the sub-tree in our ETT-nodes, replacement edges can still be found relatively quickly.

It is important to note that we only tested the threshold value of the dynamic connectivity data structure in the context of the LS algorithm, and our results might not be true for all access patterns. However, it has also been shown that it is not beneficial to push edges for more general graphs [4, pg.]. Because of this and our own analysis, we conclude that an optimised version of the algorithm should withhold from pushing edges at all and contain only a single forest in the dynamic connectivity data structure.

9.2 Recency

In [4] an optimization based on random sampling is proposed. The idea is, that when deleting an edge we randomly sample $O(\log n)$ non-tree edges as candidates for replacements before searching. Since it takes $O(\log n)$ time to examine if an edge is a replacement, this leaves the amortized $O(\log^2 n)$ complexity unchanged.

However, they also show that a deterministic enumeration of a small number of edges can improve performance in practice [4, pg.9].

We then wish to explore if we can exploit the locality of the access pattern, by storing the most recent $O(\log n)$ non-tree edges and sampling these for replacements before searching.

Let us first describe what "recency" encompasses. When growing a local separator we iteratively select a vertex to add to the separator, add all neighbours to the frontier and then remove the selected vertex from the frontier. The recency of an edge is then number of such iterations that have passed since the edge was inserted into the structure.

Before implementing this heuristic, we will determine if such a recency measure is likely to improve performance of the model, by measuring how recent the most recent replacement is.

Model	wsm0.125	wsm0.25	wsm0.5	wsm1	wsv60	wsv70	wsv 80	wsv90
Vertices	2470	4946	9898	19803	6166	9830	14723	20966
Avg. lifetime	18.22	26.73	37.66	57.34	92.03	128.09	172.82	220.67
Avg. recency	2.107	2.831	3.172	2.501	43.58	57.18	73.72	89.73
Avg. most recent	1.730	2.458	2.790	2.089	35.66	45.40	59.29	70.27

Table 2: Results of measuring recency and lifetimes

On table 2 results of measuring recency and lifetimes of edges are shown. The average lifetime denotes the average number of iterations between insertion and removal of a given edge, the average recency denotes is the average recency of the first replacement encountered, and the average most recent refers to the average recency of the most recent replacement that exists.

Since the idea for the optimisation is to store recent edges to sample before searching, the average most recent edge gives us an idea of how many iterations we would need to store edges in a buffer to ensure a reasonable hit-rate for the optimisation.

Of note is that for meshes, **wsm**, the average most recent remains low even as the number of vertices grows. It is not unreasonable to imagine that storing the most recent three iterations would increase performance on meshes, ignoring the additional computation associated with maintaining and searching through a recency-buffer.

For voxel grids, **wsv**, we would need to store many more iterations and it seems the number of iterations also grows much faster than for meshes.

Intuitively, when we are growing a sphere for selection of the next vertex to add to the seperator, it is not hard to imagine that the vertices we select in one round can be very distant from the one selected in the next round. Thus it is no surprise that we cannot generally estimate locality by this measure of recency.

Thus, we do not deem recency as a good heuristic for such an optimization in the general case.

10 Results & Discussion

From this point forward we will refer to 3 different versions of the skeletonization algorithm. The original (*old*), the improved algorithm that uses a dynamic connectivity data structure (*new*), and an optimised version of *new* that only has a single forest in its dynamic connectivity data structure (*opt*).

model	type	vertices	old	new	opt
botanical_tree	*	75849	63.61	61.60	62.53
temofoam	voxel	57880	49.42	66.15	57.30
bunny	mesh	34834	2513.65	1603.24	1228.67
neptune	mesh	28052	173.68	125.55	99.86
fertility	mesh	24994	364.91	220.26	170.86
wsv90	voxel	20966	773.94	192.12	160.90
wood_statue	mesh	19803	67.66	43.04	31.19
human_hand	mesh	12438	52.80	35.37	26.55
rotor	mesh	10663	8.78	6.37	5.51
armadillo	mesh	6488	5.93	4.25	3.14

Table 3: Runtime of each version for different models. * Model generated from point cloud.

We will inspect how the versions perform on a set of distinct models. From table 3, we immediately see that *opt* is always faster than *new*. Because *new* is both more complex and slower, ts makes little sense to spend more time on this version.

Next, we find that *opt* runs about two times faster for all meshes when compared with *old*. This seems to be the case regardless of the model and the number of vertices.

Let us examine a model that is very difficult to handle with the LS algorithm. The **bunny** model is the model that takes the longest to skeletonize regardless of algorithm. The likely reason for this is because bunny is a fairly large, almost spherical, model. This leads to large separators that take a long time to grow. Here we would like to see a larger improvement in runtime but *opt* is still only about two times faster. However, when looking at the times it takes to complete the individual steps in *opt* (table 4), we see that the time to grow separators have been massively decreased, and shrinking is now the primary factor limiting performance. Looking at similar data for the other models, we generally find that shrinking takes up more time than growing, although the difference is most noticeable with **bunny**.

	old	opt
Grow	12318	1924
Shrink	6114	6759

Table 4: The grow time and shrink time of *old* and *opt* on **bunny**. values are CPU time in seconds.

Referring to table 3 again, we see that **wsv90** sees a much larger improvement in performance between the versions. This is likely due to **wsv90** being a voxel-grid as this is the only difference between that and **wood_statue**. When speaking of voxel-grids, separators will generally grow larger and contained nodes will have more neighbours due to nodes also occupying the "inside" of the model. The result of this is that it is slower to check connectivity of clusters and performance is therefore lower. This is especially the case for the naive method used in *old*. However, *opt* seems to scale much better with this kind of graph, as evident by figure 9.

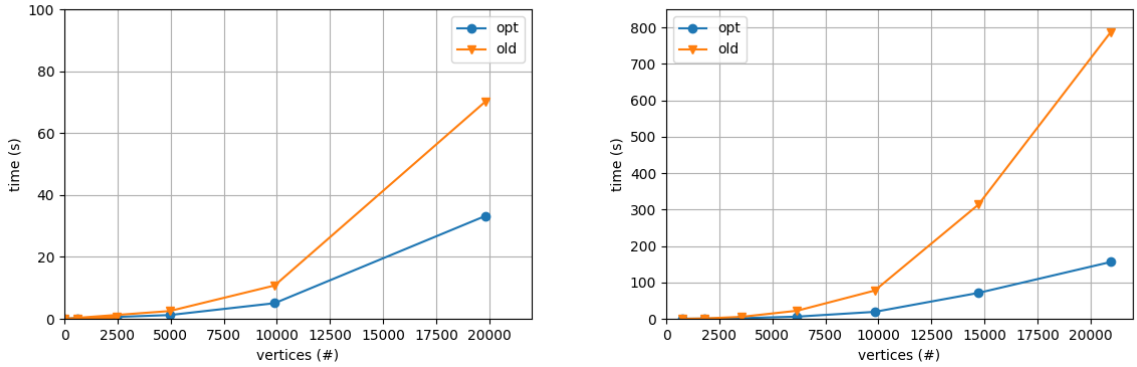


Figure 9: Runtime of *opt* and *old* on **wsm** (left) and **wsv** (right).

We can explore this further by looking at the growth time and shrinking time of *opt* on the **wsv** models. Figure 10 shows how the shrinking time grows faster than the growing time with respect to the number of vertices. In fact, the growing time seems to be growing linearly for this particular model. This again suggests that growing is no longer a bottle neck of the algorithm.

For the models **botanical_tree** and **temofoam**, we find that *opt* performs on par or even worse when compared to *old*. This seems to happen because both of these models consist of a large number of very thin "tubes". This would result in a large number of separators that are generally small. We can see this is the case because the time it takes to pack separators is larger than the time it takes to find them (figure 5). This is unlike all other models tested. Since checking connectivity in very small separators is fast, even with the naive approach used in *old*, we should not expect to see much improvement here, and for the case of **temofoam**, using a dynamic connectivity data structure even seems to be slower.

	botanical_tree	temofoam
Finding separators	20	28
Packing separators	40	30

Table 5: The time to find and pack separators of *opt* on **botanical_tree** and **temofoam**. values are wall time in seconds.

Although slower on a few models, we find that *opt* is generally an improvement over the original algorithm. This is especially true for voxel-grids where we see a substantial increase to performance.

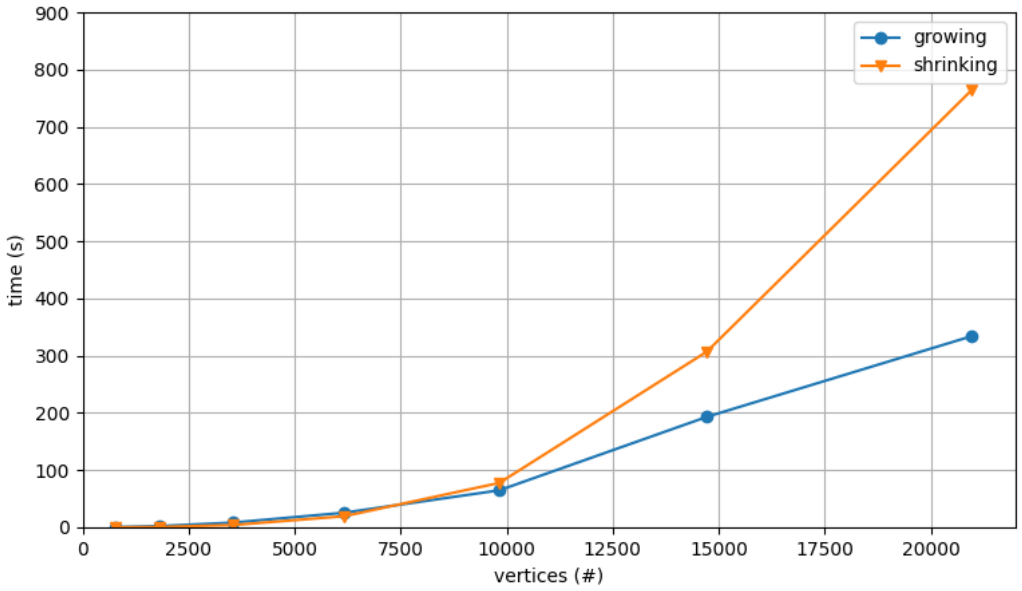


Figure 10: The CPU time to grow and shrink separators on the **wsv90** models with *opt*.

For meshes we see that *opt* runs about two times faster regardless of the model. While this means that the the algorithm will scale as well as the original for meshes, we have reached a point where checking connectivity is no longer a bottleneck and other parts of the program is what is limiting scalability.

10.1 Further Work

The scope of the project has been limited to optimising the part of local separator computation involving connectivity. As discussed in the previous section, we have managed to improve performance so that connectivity is no longer the primary bottleneck. To benefit the most from further optimisations, improvements should be focused on other parts of the algorithm such as shrinking separators, packing or otherwise reducing the number of separators computed.

We have tried to improve performance in general, but it may be possible to target optimisations to specific types of input. Input like **temofoam**, where the graphs have many small separators, has gained little or even lost performance by switching to the dynamic connectivity structure. A likely workaround to this is to use a hybrid approach that initially uses the BFS-approach to connectivity until the front has reached a certain size before switching to dynamic connectivity.

An alternative to our recency based optimisation is likely to improve performance further [4], such as one based on random sampling or one that otherwise exploits the specific access pattern of the algorithm.

11 Conclusion

The goal this project was to improved the performance of the Skeletonization with local separators algorithm. We have examined and measured what the performance bottlenecks were, and found that the dominating factor was checking connectivity. We have described theory relating to a dynamic connectivity data structure and how to integrate this structure into the algorithm. We then performed experimental evaluation of the performance of the modified algorithm, as well as exploring further practical optimisations.

Our final implementation of the algorithm is approximately two times faster on meshes and even faster on voxel-grids, with the exception of special types of input. Our improvement has made it so checking connectivity is no longer the primary factor limiting runtime. The output of the algorithm is unchanged and still of high quality.

Our findings indicate that for further improvement, optimisations will benefit more from focusing on other aspects of the algorithm than connectivity.

References

[1] Nicu D. Cornea, Deborah Silver, and Patrick Min. "Curve-Skeleton Properties, Applications, and Algorithms". In: *IEEE Transactions on Visualization and Computer Graphics* 13.3 (2007), pp. 530–548. ISSN: 1077-2626. DOI: 10.1109/TVCG.2007.1002. URL: https://doi.org/10.1109/TVCG.2007.1002.

[2] Andreas Bærentzen and Eva Rotenberg. "Skeletonization via Local Separators". In: *CoRR* abs/2007.03483 (2020). arXiv: 2007.03483. URL: https://arxiv.org/abs/2007.03483.

[3] Jacob Holm, Kristian de Lichtenberg, and Mikkel Thorup. "Poly-Logarithmic Deterministic Fully-Dynamic Algorithms for Connectivity, Minimum Spanning Tree, 2-Edge, and Biconnectivity". In: 48.4 (2001). ISSN: 0004-5411. DOI: 10.1145/502090.502095. URL: https://doi.org/10.1145/502090.502095.

[4] Raj D. Iyer et al. "An experimental study of poly-logarithmic fully-dynamic connectivity algorithms". eng. In: *Acm Journal of Experimental Algorithmics* 6 (2001), p. 4. ISSN: 10846654. DOI: 10.1145/945394.945398.

[5] Monika Henzinger and Valerie King. "Randomized Dynamic Graph Algorithms with PolyLogarithmic Time per Operation". In: *Journal of the ACM* 46 (Jan. 1995), pp. 502–516. DOI: 10.1145/225058.225269.