Polygonal Mesh Connectivity Compression using Graph Symmetries

Andrej Jočić, Uroš Čibej, Matija Marolt, Ciril Bohak, Žiga Lesar

University of Ljubljana, Faculty of Computer and Information Science, Slovenia
E-mail: aj6482@student.uni-lj.si, {uros.cibej, matija.marolt, ciril.bohak, ziga.lesar}@fri.uni-lj.si

Abstract

By treating meshes as undirected graphs and employing a graphlet dictionary, we achieve significant compression. Our method is inspired by automorphism-based compression techniques, and we explore the potential of this approach in detail. Preliminary results on a variety of sample meshes are promising and indicate that our method can substantially reduce the amount of data needed to represent complex 3D models. We evaluate the performance of the presented approach for graphlets of different sizes for compression efficiency and compression/decompression times and investigate the limitations when compressing meshes with different topologies.

1 Introduction

Polygonal models are widely used for representing 3D objects in computer graphics. With increased rendering systems capabilities and the desire for greater realism and scene precision, the size of these models has been increasing. To cut down transmission and static storage costs, specialized compression schemes have been developed. In previously established terms as presented by Magloet al. [8], this paper focuses on lossless single-rate global compression of static polygonal models. This means that we consider fixed (non-temporarily-evolving) data, which is decompressed all at once (without progressive transmission at different levels of detail) and in its entirety (not focusing on specific regions of the model as requested by the user).

The information we usually store is the model's geometry (positions of vertices), connectivity (incidence relations among elements), and optional additional attributes (normals, colors, etc.) [7]. The connectivity information typically takes up the most space – around twice as much as the vertex position information in a triangle mesh homeomorphic to a sphere [11], so it is the most important to compress. The vertex-to-vertex connectivity relations, also known as the *wireframe* W or the *1-skeleton*, can be represented as an undirected graph. Thus, we can reduce our problem to graph compression, although this loses some information, such as face orientation (see section 4).

Instead of the wireframe, we could theoretically compress the *face-to-face* adjacency graph, which is the *dual* graph of the wireframe, W^*. Assuming W is a planar graph, we could restore it at decompression by computing the dual of W^* again, since in general for a (connected) planar graph G holds the following $(G^*)^* = G$. However, this would inflate the typical face-list mesh format unless we could develop more compact face descriptors than vertex lists. If we are willing to give up the vertex-to-attribute mapping (and e.g.infer positions with a layout algorithm), we can compress the connectivity information independently from any vertex data. In this case, compressing the face incidence graph may be a viable option since we do not have to list the vertices of each face; we need an arbitrary face identifier. However, this is not something we consider in this paper since it does not appear to be a common use case in 3D graphics.

Typically, compression methods focus exclusively on manifold triangle meshes or another constrained structure, whereas we consider arbitrary polygonal models. The performance of mesh connectivity compression methods is typically measured as connectivity bit rate in terms of bits per vertex (bpv). Our main focus is higher-level data representations, with no attempt at optimal encoding at the bit level. Thus, a thorough comparison with state-of-the-art is out of the scope of this paper, but for completeness, note that Khodakovsky et al. [6] have shown a provably near-optimal encoding of polygonal manifolds achieving around 1.8 bpv on average.

1.1 Mesh File Formats

A typical mesh file format (e.g.OFF [9], PLY [12]) has a header with metadata, followed by the vertex information, and finally, the connectivity information. The connectivity is encoded as a list of faces, where each face is a list of vertex indices. Within a face, we have one vertex index per edge, but usually, most edges are shared between two faces (assuming relatively few boundary edges). This means we have about two vertex indices per edge, a baseline for storing graphs, which we aim to improve with compression. We decided to support the compression and decompression of PLY files since they are widely adopted and relatively simple in structure. Note that the method is general and not bound to the PLY format. PLY comes in two forms: compact binary and human-readable ASCII. When reporting compression rates, we consider the binary format for a fair comparison with our compressed files.

2 Graphlet-based Compression

We compress the wireframe by successively replacing graphlets (small induced subgraphs), as found by a subgraph isomorphism solver, with their more compact representations. Our algorithm follows the structure of [1, Algorithm 1], except for the graphlet encoding scheme. Whereas Čibej and Mihelič used their so-called *symmetry-compressed* graph representation, we opted for a simple dictionary-based approach.

A graph G is defined to be *symmetry-compressible* (SC) if it can be more succinctly represented with a *residual* graph G^π and an automorphism π of G. Here, G^π is the minimal set of edges required to reconstruct G by applying π to its edges until all permutation cycles are closed. The compression procedure loops through a precomputed list of SC graphlets with $n \leq n_{\max}$ nodes. For each graphlet G', it greedily extracts a maximal edge-disjoint set of subgraphs isomorphic to G', to be replaced by their SC representations. As a heuristic, the graphlets are first sorted in decreasing order of relative compression efficiency to make sure that highly compressible ones are found before any of their edges have already been removed.

Instead of encoding a subgraph $H \cong G'$ with (π, H^π), we opted to store the index of G' in a *graph atlas* [10] (a systematic enumeration of all undirected graphs up to a certain number of nodes $n_{\max}$) along with a mapping between nodes of H and G' (which have an agreed-upon order within each G'). Taking only $1 + n$ integers, this is always at least as efficient as the SC representation of G' since encoding of π alone requires at least n integers (barring a much more efficient permutation encoding than the one proposed in [1]). Thus, there is no reason to combine the representations, using the atlas for some graphlets and symmetry compression for others.

This approach may be seen as a sort of *instancing* on a sub-mesh level. A graphlet instance refers to an isomorphic version of itself in the atlas, but for each, we have to store a separate vertex mapping (analogous to the coordinate transform in the case of classical instancing). This improvement requires an agreed-upon dictionary between the encoder and decoder – atlas, which is unnecessary in symmetry compression, plus the auxiliary space the atlas takes in memory. We used the 1253 graphlets with $n \leq 7$ as provided by Python's `networkx` library [3]. This size limitation is not much of a practical concern, as searching for even larger subgraphs results in prohibitively long compression times.

We define the relative efficiency of an atlas-based encoding as

$$\delta^r(G) = 1 - \frac{1+n}{2m}, \tag{1}$$

for a graphlet G on n nodes with m edges (analogous to the same definition for an SC encoding [1, Section 4]), which may be used for sorting template graphlets before compression.

Let us define *atlas-compressible* (AC) graphs as ones that can be encoded this way more compactly, meaning $\delta^r(G) \in (0, 1)$. While most connected graphlets with up to 9 nodes are SC [1, Table 2], many are not. This includes the most common structure found in polygonal meshes, the triangle [1, Theorem 3]). The good news is that the triangle has $\delta^r(K_3) > 0$, meaning it *is* atlas-compressible. Even better, it turns out that *all* connected graphs with $n \geq 4$ are AC.

Proof: We must show $\frac{1+n}{2m} < 1$ for $n \geq 4$. For a connected graph we have $m \geq n - 1$, so $\frac{1+n}{2m} \leq \frac{m+2}{2m} < \frac{m+3}{m+m} \leq 1$ since $m \geq n - 1 \geq 3$ for $n \geq 4$.

2.1 Compressed file format

The compressed mesh is serialized to a binary file as follows. First, the header and vertex data are copied from the original PLY into their own file. Next, the atlas-compressed connectivity data is written to another file as a series of 4-byte unsigned integers. First come the residual (uncompressed) edges as pairs of vertex indices, preceded by the actual number of such edges. Finally, AC representations of the compressed subgraphs are stored. The most direct encoding would store the atlas index and node mapping for each subgraph independently (note that we can infer the size of a graphlet, and thus the length of the mapping, from its index). But this approach duplicates the same atlas index many times if graphlets often reoccur in the mesh, which indeed we observed on the meshes we tested. So instead we used an index *multiplicity* encoding, where each *unique* atlas index is stored along with the number of times it occurs in the mesh, followed by this many vertex mappings.

In theory, this encoding is better than the direct one if the total number of compressed graphlets is more than twice the number of unique atlas indices; in other words, if the average graphlet frequency is more than 2. This threshold was significantly exceeded in all our experiments, so we use this encoding unconditionally. Otherwise, we could have used a hybrid approach, where we fall back to the direct encoding if the average index multiplicity is too low. This requires an extra bit in the binary to signal how it should be decoded.

Note that 4 bytes are excessive for some of these integers, certainly the atlas indices, which only go up to 1252 when $n_{\max} = 7$. We also precede the atlas-encoded file segment with the *number* of unique indices (which is technically redundant) for the purposes of convenient decoding. We made no attempt at the most efficient binary encoding possible, leaving this for future work.

2.2 Experiments

We have tested our implementation on a set of sample meshes provided with MeshLab (available at `meshlab.net/#download`). Table 1 shows compression performance on `non_manif_hole.ply`, a non-manifold mesh with 755 triangles. We defined relative compression efficiency over the whole wireframe W as the relative decrease in the number of integers needed for our encoding (Section 2.1) compared to the baseline of $2m$ for a mesh containing m edges. Just as equation 1, this is a value from 0 to 1 where higher is better. The compression time was recorded with a pure Python implementation running on a Ryzen 7 5800X3D. Evidently, having

Table 1: Compression performance for model `non_manif_hole.ply` with different $n_{\max}$ (maximum graphlet size). Relative efficiency δ^r is computed over the whole wireframe, R_{conn} is the compressed-to-original size ratio for the connectivity data, R_{total} is the ratio for the entire PLY file.

$\mathbf{n}_{\max}$	$\delta^{\mathbf{r}}(\mathbf{W})$	$\mathbf{R}_{\mathbf{conn}}$ **(%)**	$\mathbf{R}_{\mathbf{total}}$ **(%)**	**time (s)**
4	0.534	45.0	64.2	0.4
5	0.564	42.1	62.3	2
6	0.596	39.0	60.3	17
7	0.633	35.4	57.9	185

Table 2: Compression performance on more sample meshes for compression metrics reported with $n_{\max} = 4$ and $n_{\max} = 5$.

name	**triangles**	$\delta^{\mathbf{r}}(\mathbf{W})$		**time (s)**	
		$\mathbf{n}_{\max} = \mathbf{4}$	$\mathbf{n}_{\max} = \mathbf{5}$	$\mathbf{n}_{\max} = \mathbf{4}$	$\mathbf{n}_{\max} = \mathbf{5}$
bunny2	1000	0.522	0.576	0.5	3
bunny10k	9999	0.531	0.584	4	39
bunny70k	69451	0.537	0.580	31	306
screwdriver	13574	0.534	0.581	7	59

access to graphlets with sizes up to 7 is sufficient because compression time is already approaching the limits of practicality.

Table 2 shows compression performance on other MeshLab sample meshes. We have only shown the relative efficiency since this tells us how appropriate atlas-based compression is for polygonal models. Raw compression ratios are not listed, as they are not so informative given our inefficient binary serialization and the fact that we left vertex data entirely uncompressed.

3 Bipartite Subgraph Completion

We explored the use of bipartite subgraph completion for mesh wireframe compression. The algorithm is based on the slightly extended notion of *near symmetry-compressible* (NSC) graphs, which can be made SC by removing and/or adding a few edges. The idea is to repeatedly find near-complete bipartite subgraphs of G (which have high relative efficiency) and replace them with their NSC representation. These are found using a greedy optimization starting from each graph vertex.

Though this algorithm generally runs faster than the isomorphism matching approach described previously, unfortunately, it yields comparatively poor relative efficiency (around 0.1 to 0.15 for the meshes we tested), so we did not pursue this direction further. These results may be explained by the fact that typical polygonal meshes are planar or locally plane-like (they have Euler characteristics close to 2) since they model object surfaces. A complete bipartite graph $K_{n,m}$ with $n \geq 3$ and $m \geq 3$ cannot be planar according to Wagner's theorem. Therefore, the only complete bipartite subgraphs that may frequently appear in polygonal models are $K_{1,m}$ (a star), and $K_{2,m}$, but the former is not NSC. The vast majority of extracted subgraphs we observed in meshes were $K_{2,2}$, i.e. a 4-cycle. The NSC representation of this graphlet only has relative efficiency 0.125, which explains the poor global efficiency ratios.

4 Mesh Reconstruction

When it comes to reconstructing the original mesh, decompressing the wireframe is hardly the end of the story. To produce a list of faces (as in the original file), we need to compute the facial walks of the wireframe. In the case of a regular mesh (with all faces having the same degree), we can enumerate all simple cycles of this length. This has time complexity $O((f+n)(d-1)k^d)$ for a graph with n vertices, f faces of degree d, and average vertex degree k. Even though $k < 6$ for planar graphs, this can still take quite a while for large meshes. For the 10,000 triangle bunny mesh listed in Table 2, this took 3 entire minutes. Otherwise, if we are enumerating cycles of varying degrees, we need to ensure that all faces of non-minimal degree are *chordless*, i.e. they do not have any diagonals. This even further increases decompression time. Note that our tool does not currently support decompression of such non-regular meshes.

There is also the issue of *internal* cycles, whose corresponding faces would end up inside the mesh (just under the surface). In practice, virtually all meshes represent *surfaces*. Therefore, it seems reasonable to assume these were not present in the original mesh. Going with this assumption, we identify all such false positives as the faces with all edges incident on more than two total faces and remove them at decompression. Ideally, our tool would detect such edge cases and encode them explicitly in a header of the compressed file instead of making assumptions.

The facial walks also lack a crucial piece of information: face orientation. We can think of this as a clockwise or counter-clockwise vertex ordering. This information is needed by renderers to compute the surface normals (for shading etc.) and to decide which faces are front-facing. We are faced with a trade-off: either we put the burden of computing proper orientations on the system rendering the mesh, or we reduce the utility of the compressor by explicitly storing them (inflating space) or somehow computing them at decompression (increasing its runtime). We have observed that Blender 4.0 can fix inconsistent face orientations automatically, but MeshLab 2023.12 did not attempt to do so, resulting in incorrect shading on about half of the faces if we do not address the issue.

We have only implemented a solution for orientable triangle meshes, where we can arbitrarily fix the orientation of one face (on each connected component) and propagate it to the rest of the mesh without producing inconsistencies. While this could be easily extended to general orientable manifolds, we would need a different method for all other meshes. A reasonable option might be using a point-cloud normal estimation technique (such as [4, Section 3.3]) on the underlying vertices.

If we wanted to preserve orientations exactly as they were in the original file, we could append them to the compressed binary as a bit-vector, but this requires a reproducible ordering of faces (e.g., rank in lexicographic ordering). This is a worthwhile direction to pursue in the future since such an ordering would allow us to carry over

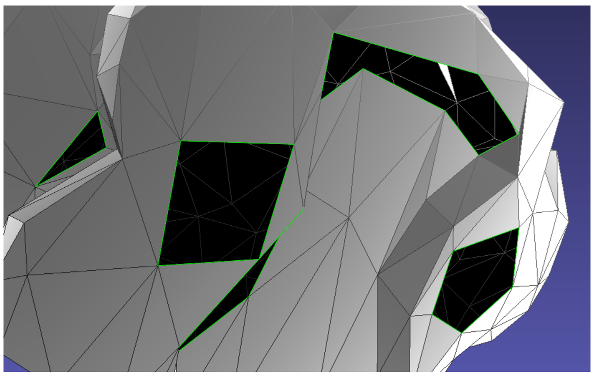

Figure 1: Holes at the base of the bunny2.ply mesh. The leftmost one, consisting of a single triangle, cannot be inferred from the wireframe alone. The rest are kept in the reconstructed mesh since their boundaries are longer than 3, while the single-face hole is erroneously patched up.

arbitrary face data (like colors) that can appear in PLY files. Devising a reproducible ordering of faces comes with the issue of single-face holes (see Fig. 1). Since these cannot possibly be inferred from the wireframe alone, we would need to find them before compressing the mesh and encoding them explicitly. To that end, we could enumerate all cycles in the wireframe and check which are not present as faces in the original mesh. However, our existing face enumerator would significantly increase compression time. This is such a niche case that we have simply ignored for now, which results in such holes being patched up in the decompressed mesh.

5 Conclusion

We have implemented a prototype PLY-format mesh compression tool [5] which can compress the connectivity data of a moderately sized mesh to around 45% of its original size in a reasonable time. Aside from inefficient serialization (Section 2.1) and reconstruction limitations (Sec. 4), there are still many things to work on before it becomes fit for general use.

The class of near-symmetry compressible graphs mentioned in section 3 is a general formulation from which we could derive many domain-specific algorithms. Unfortunately, the bipartite completion algorithm turns out to be unsuited for the graphics domain, but we could develop something more tailored to the structure of polygonal meshes. For example, if we limit ourselves to manifolds or, even further, to triangular or quadrangular manifolds, we may be able to find highly compressive symmetries. Alternatively, the assumption of a (near-)planar graph may allow us to speed up certain computations like facial walk enumeration and reduce the search space for compressible features (e.g., filter out non-planar graphlets in a matching algorithm). The heuristic sorting of graphlets could also account for common graphlet frequencies in this domain, which we have observed to be very skewed. A more informed heuristics may allow us to break out of the loop over graphlets early (based on a heuristic value threshold) without severely impacting the final compression ratio. This way, we could consider even larger graphlets that can be (potentially) more dense, which increases the efficiency (1) of their AC representation.

A more comprehensive evaluation of our compressor is required. Since the efficacy of this kind of compression is very much dependent on the density of the mesh, we would like to observe performance metrics *with respect to* mesh density. To make these metrics comparable, we should use meshes of the same object with different levels of tessellation. An appropriate tool for generating such a series from a point cloud might, for example, be a Čech filtration [13, p. 62].

Our approach is limited to the simplest case of lossless single-rate global compression of static data. However, it may naturally be extended to other compression modalities. For example, lossy compression could be achieved by replacing graphlets with similar ones that can be represented more compactly based on human perception. Such perceptual metrics have been studied in the literature [2].

References

[1] Uroš Čibej and Jurij Mihelič. Graph automorphisms for compression. *Open Computer Science*, 11(1):51–59, 2021.

[2] Massimiliano Corsini, Mohamed-Chaker Larabi, Guillaume Lavoué, Oldřich Petřík, Libor Váša, and Kai Wang. Perceptual metrics for static and dynamic triangle meshes. *Computer graphics forum*, 32(1):101–125, 2013.

[3] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. Exploring network structure, dynamics, and function using networkx. In Gaël Varoquaux, Travis Vaught, and Jarrod Millman, editors, *Proceedings of the 7th Python in Science Conference*, pages 11 – 15, Pasadena, CA USA, 2008.

[4] Hugues Hoppe, Tony DeRose, Tom Duchamp, John McDonald, and Werner Stuetzle. Surface reconstruction from unorganized points. In *Proceedings of the 19th annual conference on computer graphics and interactive techniques*, pages 71–78, 1992.

[5] Andrej Jočić, Uroš Čibej, Matija Marolt, Ciril Bohak, and Žiga Lesar. Polygonal Model Compression with Graph Symmetries. https://github.com/UL-FRI-LGM/mesh-compressor. [Online; accessed 5-July-2024].

[6] Andrei Khodakovsky, Pierre Alliez, Mathieu Desbrun, and Peter Schröder. Near-optimal connectivity encoding of 2-manifold polygon meshes. *Graphical Models*, 64(3-4):147–168, 2002.

[7] Žiga Lesar and Matija Marolt. Graphs in computer graphics. *Uporabna informatika*, 31(2), aug 2023.

[8] Adrien Maglo, Guillaume Lavoué, Florent Dupont, and Céline Hudelot. 3d mesh compression: Survey, comparisons, and emerging trends. *ACM Comput. Surv.*, 47(3), feb 2015.

[9] Object File Format. https://segeval.cs.princeton.edu/public/off_format.html. [Online; accessed 5-July-2024].

[10] Ronald C Read and Robin J Wilson. *An atlas of graphs*. Oxford University Press, 1998.

[11] Jarek Rossignac. Edgebreaker: Connectivity compression for triangle meshes. *IEEE transactions on visualization and computer graphics*, 5(1):47–61, 1999.

[12] Greg Turk. The PLY Polygon File Format. https://web.archive.org/web/20161204152348/http://www.dcs.ed.ac.uk/teaching/cs4/www/graphics/Web/ply.html. [Online; accessed 5-July-2024].

[13] Žiga Virk. *Introduction to Persistent Homology*. Založba UL FRI, 2022.