Early-Split Coding of Triangle Mesh Connectivity

Martin Isenburg
UC Berkeley
isenburg@cs.berkeley.edu

Jack Snoeyink
UNC Chapel Hill
snoeyink@cs.unc.edu

ABSTRACT

The two main schemes for coding triangle mesh connectivity traverse a mesh with similar region-growing operations. Rossignac's Edgebreaker uses triangle labels to encode the traversal whereas the coder of Touma and Gotsman uses vertex degrees. Although both schemes are guided by the same spiraling spanning tree, they process triangles in a different order, making it difficult to understand their similarities and to explain their varying compression success.

We describe a coding scheme that can operate like a label-based coder similar to Edgebreaker or like a degree-based coder similar to the TG coder. In either mode our coder processes vertices and triangles in the same order by performing the so-called "split operations" earlier than previous schemes. The main insights offered by this unified view are (a) that compression rates depend mainly on the choice of decoding strategy and less on whether labels or degrees are used and (b) how to do degree coding without storing "split" offsets. Furthermore we describe a new heuristic that allows the TG coder's bit-rates to drop below the vertex degree entropy.

CR Categories: I.3.5 [Computational Geometry and Object Modeling]: Boundary representations

Keywords: Connectivity compression, label coding, degree coding, split offsets, Edgebreaker, TG coder, Cut-border Machine.

1 INTRODUCTION

A number of schemes for coding the connectivity of triangular meshes have been proposed [3, 11, 14]. All these schemes grow a region on the mesh by including triangle after triangle into a boundary and encode a sequence of symbols that documents this process. The Edgebreaker scheme [11] and the Cut-border Machine [3] are *label-based* approaches. They encode for each triangle how its inclusion changes the boundary using one of five labels. The TG coder [14] is a *degree-based* approach. It only encodes a symbol if including the triangle reaches a new vertex, in which case the degree of this vertex is encoded, or if including the triangle splits the boundary into two loops, in which case a "split" symbol is encoded.

Although label-based and degree-based schemes perform nearly identical region-growing traversals, it is difficult to compare the encodings they produce. While they traverse the vertices along the same spiraling spanning tree, they process the triangles in a slightly different order, making it impossible to establish a one-to-one mapping between label-based and degree-based encodings. This has prevented a deeper understanding of what exactly makes one algorithm compress better than another. In this paper we describe a unifying coding scheme that can produce *either* a label *or* a degree encoding thereby allowing a direct comparison between the two.

The lack of insight how label and degree coding are different and how they are similar has also hindered the design of a degree coder that avoids storing explicit "split offsets." It was speculated that it would be possible to modify the TG coder to operate without explicitly storing those integer offsets that are associated with every "split" symbol and that specify the size of the boundary loop that gets "split off." Such speculations come from the fact that Edgebreaker manages to avoid storing "split offsets," whereas the otherwise identical Cut-border Machine stores them explicitly.

Edgebreaker can recover all "split offsets" by performing an additional pass over the labels [11] or the mesh [12], or by processing the label sequence in reverse [6]. For each "split" label S, Edgebreaker has a corresponding "end" label E that delimits a label subsequence, which implicitly encodes these offsets. The TG coder does not store "end" symbols that would allow to identify similar degrees subsequences. However, we have shown that simply adding "end" symbols is not sufficient to make offsets redundant [8].

Edgebreaker and the Cut-border Machine perform significantly more boundary splits than the TG coder. They occasionally leave behind *warts*, unprocessed triangles that share two edges with the boundary, that cause additional splits later. Knowledge about these additional splits constitutes extra information in the label sequence that is available to Edgebreaker for recovering split offsets but that is missing in the degree sequence of the TG coder. This observation serves as further evidence *that* we need to add extra information beyond "end" symbols for making split offsets implicit to the symbol sequence of a degree coder. In this paper we describe a scheme that offers a unified view on label-based and degree-based coding and makes it easy to see *what* this extra information must be.

The connectivity coding scheme detailed in the following can produce either labels or degrees. In either mode it performs the exact same boundary splits and processes vertices and triangles in the exact same order. This is achieved (a) by performing "split" operations earlier than the TG coder and Edgebreaker and (b) by not leaving behind warts. To immediately remove warts that have formed on the boundary we use a new "wart" operation. It turns out that the extra information needed for making split offsets implicit is the knowledge about when this "wart" operation was used.

Our *early-split coder* can operate similar to Edgebreaker using one label per triangle without storing offsets, but also similar to the TG coder using one degree per vertex and explicit offsets. Most importantly, it allows degree coding without offsets by adding "end" and "wart" symbols to the degree sequence. In fact, our coder can be made to produce one of six possible symbol sequences: three label-based and three degree-based. The first of each triple is a sequence that contains explicit split offsets and is decoded in a single forward pass. Here the label-based sequence corresponds to the Cut-border Machine [3] and the degree-based sequence to the TG coder [14]. The second sequence does not contain explicit offsets and can be decoded in two passes. Here the label-based sequence corresponds to Edgebreaker [11] and the degree-based sequence is new. The third sequence does not contain explicit offsets and can be decoded in one reverse pass. The label-based sequence corresponds to Spirale Reversi [6] and the degree-based version is also new.

We report representative compression rates for all six approaches. The results suggest that for maximal compression it is less important whether an encoding uses label or degrees. What matters most is whether decoding is done forward with offsets, forward without offsets, or in reverse. Forward decoding without offsets consistently gives the worst compression rates. Because decoding is done in two passes the decoder has relatively little context for predicting the next symbol whereas the other decoders can base their predictions on a partially reconstructed mesh. The best compression is always achieved by reverse decoding, while forward decoding with offsets is a close second. For the latter, which is of particular interest because it allow one-pass encoding and decoding, using degrees was always the better than using labels—mainly because it allows employing a novel heuristic.

Our final contribution is this novel heuristic that improves compression rates of degree-based forward decoding with offsets. In particular, this heuristic can also be used with the original TG coder, allowing it—for the first time—to achieve compression rates below the entropy of the vertex degree sequence. Previously this had only been achieved by reverse decoding [13]. Despite its simplicity, our heuristic gives bit-rates that beat those achieved with the much more complex adaptive-conquest heuristic of Alliez and Desbrun [1], which is still bound by the vertex degree entropy.

2 TG Coder – Cut-border Machine – Edgebreaker

Independently developed, the TG coder [14], the Cut-border Machine [3], and Edgebreaker [11] perform a nearly identical region-growing process to encode a mesh. The schemes maintain a compression boundary into which they include triangle after triangle. While all three coders process the vertices in the same order, the TG coder processes the triangles in a slightly different order than the other two. Edgebreaker and the Cut-border Machine always include the triangle that is adjacent to the *gate* edge, which advances in clockwise order around the *focus* vertex (see Figure 1). For the most part the TG coder does the same. However, it will immediately include any triangle that shares two edges with the compression boundary, even if neither of these two edges is the gate. The Cut-border Machine and Edgebreaker allow such triangles, which Rossignac has termed *warts* [10], to remain on the boundary. As we want our coder to process triangles in the same order—in both label and degree mode—we will not allow it to leave warts behind.

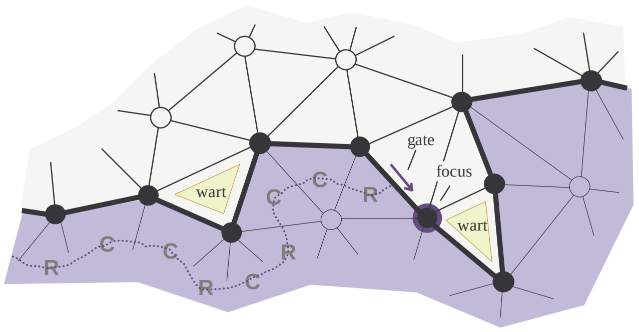


Figure 1: Edgebreaker and the Cut-border Machine always include the triangle at the gate into the boundary. Occasionally this leaves behind triangles that share two edges with the boundary (warts) that cause additional “splits” later.

Including a triangle that shares only one edge with the compression boundary (i.e. the gate) usually makes this triangle’s third vertex a new boundary vertex. Occasionally, however, this third vertex will already be on the boundary elsewhere. In this case the boundary splits into two loops, one of which is temporarily stored on a stack while encoding continues on the other. All three coding schemes perform these “split operations,” but they occur much more frequently for the Cut-border Machine and Edgebreaker. The reason is those warts left behind along the boundary that eventually lead to additional splits (see Figure 1). For the TG coder a “split operation” only occurs if both resulting boundary loops still enclose unprocessed vertices. Each of these “splits” also occurs for the Cut-border Machine and Edgebreaker. But in addition these two coders have many split operations that merely split off a wart.

The only real difference between the Cut-border Machine and Edgebreaker is that the Cut-border Machine stores an explicit offset with the symbol that encodes a “split” operation whereas Edgebreaker does not. This offset specifies the number of vertices in clockwise or counterclockwise direction along the boundary that are between the gate and the third vertex of the triangle causing the split. Having these explicit offsets enables the decoder of the Cut-border Machine to replay the encoding process. The decoder of Edgebreaker, on the other hand, needs to perform two passes or operate in reverse to recover the offsets, which are implicitly stored in label subsequences and are therefore in some sense redundant.

The TG coder also stores an explicit offset for every split operation. For quite some time it was not clear whether it would also be possible to omit these offsets. Yet, the TG coder does not store explicit “end” symbols that act as the delimiters for the symbol subsequences from which Edgebreaker derives each split offset. This seemed to suggest that the TG coder would at least need to store “end” symbols in order to operate without offsets. However, we have shown that simply adding “end” symbols to the degree sequence is not yet sufficient to make the split offsets redundant [8].

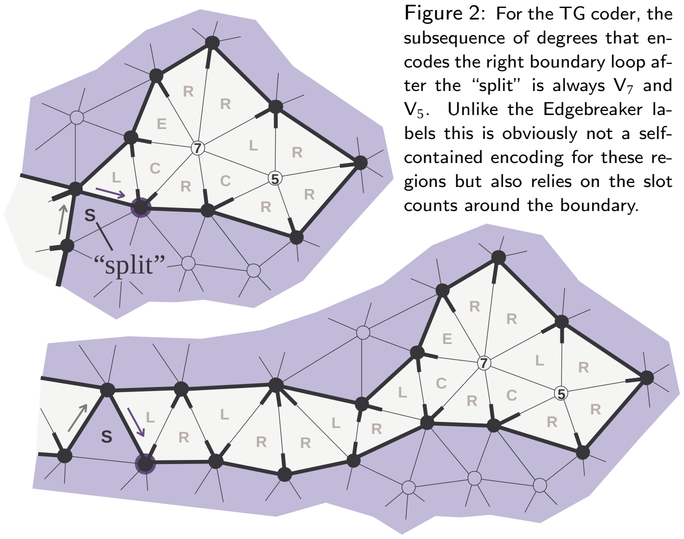
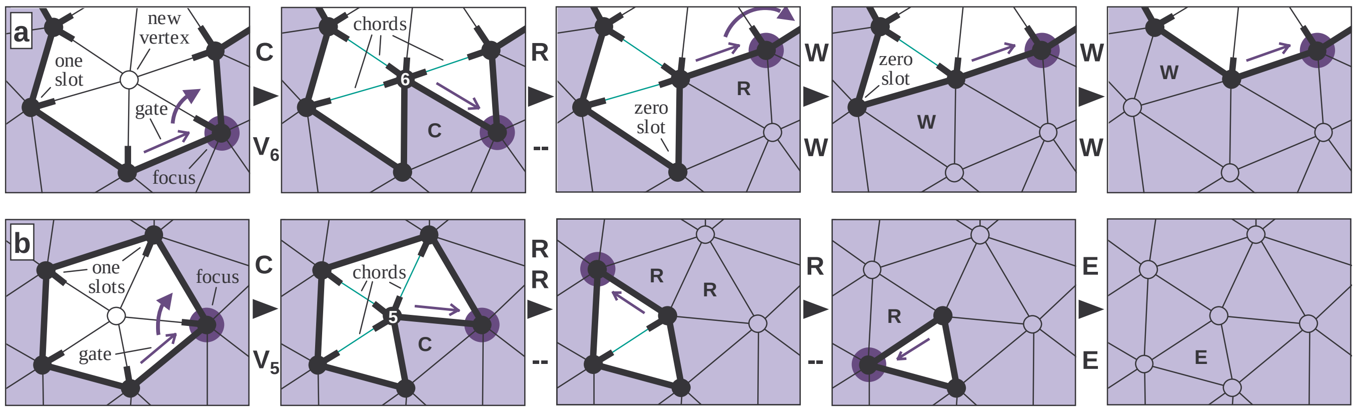


Figure 2: For the TG coder, the subsequence of degrees that encodes the right boundary loop after the “split” is always V_7 and V_5. Unlike the Edgebreaker labels this is obviously not a self-contained encoding for these regions but also relies on the slot counts around the boundary.

Unlike the label subsequences of Edgebreaker, the degree subsequences of the TG coder are not self-contained encodings of some portion of a mesh. Decoding also depends on the state of the compression boundary. The TG coder maintains much more state information than Edgebreaker keeping *slot counts* that specify how many unprocessed edges are incident to each boundary vertex. These slot counts enable the TG coder to omit explicit “end” symbols: The completion of a boundary loop is detected as the moment that all slot counts are zero. They also enable the TG coder to detect and remove warts without an explicit symbol: A wart is detected when the slot count at some vertex other than the focus drops to zero. The value and the order of the slot counts around the boundary depend on all preceding symbols. Therefore it is impossible to derive split offsets solely from subsequences of degrees (see Figure 2).

To have our coder produce self-contained degree subsequences we must keep boundary loops completely free of *zero-slots*. This requires removing zero-slots that correspond to warts left behind on the boundary as in Figure 1, as well as avoiding zero-slot creation when splitting the boundary as in Figure 2. To remove warts when operating like the TG coder we simply check for zero-slots. To remove warts when operating like Edgebreaker we store a new “wart” label W that explicitly tells us when to remove a wart. To avoid creating zero-slots when splitting the boundary we “split” earlier, before zero-slots can form. The new “wart” symbol corresponds to the extra information we need to store in addition to “end” symbols to have degree subsequences be self-contained encodings and make split offsets implicit. As for Edgebreaker, the split offsets can then be recovered either in two passes or in a single reverse pass.

We should mention that there are several incentives to include explicit offsets in the encoding. They allow decoding in a single forward pass over the symbol sequence, which makes it possible to



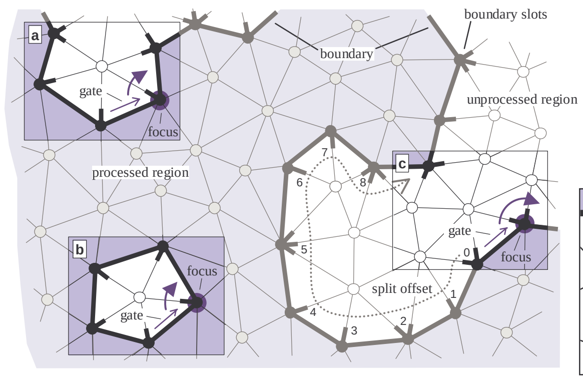
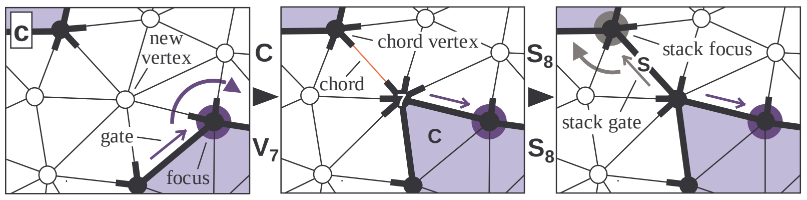


Figure 3: Three frame sequences that illustrate the scenarios that can arise during encoding. **a)** The C operation creates three chords that are removed with one R and two W operations. For degree coding the W symbols are only needed if no split offsets are stored, otherwise the degree V_6 alone is sufficient. **b)** The C operation creates three chords that are removed with three R and one E operation. **c)** The C operation creates one chord that is removed with an S operation. The offset that (only for replay decoding) is stored explicitly with each "split" specifies the number of boundary vertices between the new vertex and the chord vertex.



decompress in a streaming fashion [4]. Furthermore storing split offsets allows non-recursive mesh traversals. Moving the focus in a breadth-first rather than a depth-first manner, for example, leads to more coherent mesh layouts [5]. While the Cut-border Machine and the TG coder can easily be modified to operate this way, Edgebreaker cannot because it requires a recursive traversal in order for the omitted offsets to be implicit to the symbol sequence.

3 Encoding

Our early-split coder does two things differently from Edgebreaker and the TG coder. It performs "split" operations earlier and it immediately removes "warts." These two modifications allow our coder to be either label-based or degree-based, yet still traverse triangles in the exact the same way, providing us with a one-to-one mapping between label and degree coding and the insights that come along with this. As we describe our encoding scheme, we point out where it is identical and where it is different from previous schemes.

Like most schemes, our encoder grows a region on the connectivity graph of the mesh by maintaining one or more compression boundaries. It uses five different operations to update these boundaries. We refer to them as C, R, W, S, and E operations and also as *add*, *right*, *wart*, *split*, and *end* operations—loosely following the nomenclatures of Rossignac [11] and Touma and Gotsman [14].

The encoder starts with an initial compression boundary of length two that is defined around an arbitrary edge of the mesh. One of the two boundary edges becomes the *gate*, its target vertex becomes the *focus* and its origin vertex becomes the *pre-focus*. Then the encoder includes triangle after triangle into the boundary.

In the common case the encoder includes the triangle adjacent to the gate, which either adds a new vertex to the boundary or completes the focus. In some situations, however, the encoder either includes a triangle that is not adjacent to the gate or splits the current boundary into two loops without including a triangle. It does this to remove so-called *chord edges* that have formed on the boundary. The *immediate* removal of these *chords* is the main difference between our early-split coder and Edgebreaker or the TG coder.

A chord is an edge of the unprocessed mesh region that connects two vertices of the boundary. Such chord edges can form whenever a *new vertex* is included into the compression boundary (i.e. after an "add" operation in [14], after a "new vertex" operation in [3], or after a "C" operation in [11]). A chord always connects this new vertex to some other vertex on the boundary called the *chord vertex*.

We distinguish three types of chord edges and consequently have three different operations for removing them: The R or *right* operation and the W or *wart* operation remove a chord by including a triangle into the boundary. The S or *split* operation removes a chord by splitting the boundary into two. In Figure 3 we show example situations that demonstrate how chord edges of the three types can form, together with the operations that are used for removing them.

The R or *right* operation is used when a chord connects the new vertex to the next possible vertex in counterclockwise direction along the boundary. Such a chord separates the triangle at the gate from the remaining unprocessed region of the mesh. We include this triangle and move the focus to the chord vertex. This operation corresponds to the "connect-forward" operation from [3] and the R operation from [11]. The TG coder implicitly performs this operation whenever the slot count at the focus drops to zero.

The W or *wart* operation is used when a chord connects the new vertex to the next possible vertex in clockwise direction. Such a chord also separates a triangle from the unprocessed region. We include this triangle but focus and gate remain where they are. Neither the Cut-border Machine nor Edgebreaker perform wart operations. They ignore these type of chords, leaving behind "warts" on the boundary. The TG coder, however, also performs wart operations whenever the corresponding slot count drops to zero.

The S or *split* operation is used when a chord connects the new vertex to some other vertex on the boundary. Such a chord separates the unprocessed region into two unprocessed regions that both contain unprocessed vertices. Unlike the right and the wart operation, the split operation does not include a triangle. It merely splits the compression boundary into two loops, one of which is pushed onto a stack for later processing. The chord vertex and the chord become the focus and the gate for the boundary pushed onto the stack and we continue with the current focus and the current gate.

Our split operation is quite different from the split operation performed by Edgebreaker, the TG coder, and the Cut-border Machine. The rationale behind the re-design of the split operation is the same as the rationale for introducing the wart operation: we want the boundary loops to remain free of zero-slots so that the two symbol subsequences that encode the two unprocessed regions resulting from a split are self-contained encodings of these regions, both for the label-based and for the degree-based sequences.

Chords are removed by performing R operations before W operations and finally S operations. For the latter we start with the first chord that is found when searching clockwise around the new vertex. This gives the invariant that the boundary is free of zero-slots (and free of chords) if the next operation is to include a new vertex. Similarly, it guarantees that all boundaries on the stack are zero-slot free (and chord free), which assures that the first operation after popping a boundary from the stack is always a C operation.

For label-based coding we store the labels of all operations. For the degree-based coding we only store the degree of newly added vertices and either only split symbols (but also their associated offsets) or split, wart, and end symbols (but no offsets). The first two vertex degrees are encoded on initialization. If we store offsets they correspond to the number of vertices k that are in clockwise direction between the new vertex and the chord vertex (see Figure 3). For the degree-based decoder we encode in addition which slot the chord connects to at the new vertex and at the chord vertex. More exactly, we encode how many of the slots at these vertices move over to the boundary part that is pushed onto the stack. At each vertex there must be at least one slot that moves over, otherwise the stack boundary would have a chord, which violates our invariant.

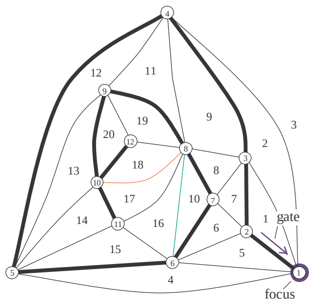
This describes our encoding algorithm for the case of a closed, single-component, genus-zero triangle mesh. Such a mesh is homeomorphic to a planar triangulation. The necessary extensions for dealing with general meshes containing holes and/or handles are similar to those used in other schemes [14, 3]. Following the example in Figure 5 should make it easy to implement this algorithm.

4 Decoding

The early-split coder can produce six different symbol sequences that encode the connectivity—three of them are label-based and three of them are degree-based. Hence, there are three decoding methods that consume a label sequence and three that consume a degree sequence. They either operate *forward with offsets*, *forward without offsets*, or in *reverse*. The reverse decoder requires the coder to perform a second pass over the produced symbols for compressing them in reverse order. It also operates without offsets.

The six different symbol sequences are illustrated in Figure 4 for a small example triangulation. For decoding from the forward sequences with offsets we perform an exact replay of the encoding process. For decoding from the forward sequences without offsets we first create a special vertex spanning tree from which the triangulation is reconstructed in a second step. For decoding from the reverse sequences we perform the boundary updates that have happened during encoding in the exact reverse order.

The remainder of this section provides details that are important for implementing the six different decoding methods. To gain a quick understanding of how these decoding algorithms work the reader may skip this section for now and first follow the examples in Figures 5, 6, and 7 where we give an annotated step-by-step illustration of reconstructing the mesh shown in Figure 4 from all six symbol sequences. Coming back to this section later will be helpful for learning how we compress the symbols into a bit-stream. From a more theoretical point of view, some readers may also find the method of decoding through *closure* of a *leaf-tree* interesting.



forward with offset:

C C C C R C R C R C R W C R C S_1 C R R E C R E

V_5 V_4 V_5 V_5 V_6 V_6 V_4 V_8 V_5 V_5 $S_{1,0,0}$

forward without offsets:

C C C C R C R C R C R W C R C S C R R E C R E

V_5 V_4 V_5 V_5 V_6 V_6 V_4 V_8 W V_5 V_5 S V_4 E V_3 E

reverse:

E R C E R R C S C R C W R C R C R C R C C C C

E V_3 E V_4 S V_5 V_5 W V_8 V_4 V_6 V_6 V_5 V_5

Figure 4: Starting from the gate edge the early-split coder processes vertices and triangle in the enumerated order. It can produce either of the six symbol sequences shown on the right. They correspond to three different decoding methods that are either label-based (upper line) or degree-based (lower line).

4.1 Forward with Offsets

For forward sequences with explicit split offsets the decoding process is an exact replay of the encoding process. For the label-based sequence, our decoder is similar to the Cut-border Machine [3] and implements ideas from Gumhold [2] and Szymczak [13] to improve compression of the labels. For the degree-based sequence, our decoder is similar to the TG coder [14] but includes a novel heuristic that allows us to further improve the compression. This makes it for the first time possible to compress the degree sequence with fewer bits than the entropy of the vertex degree distribution dictates.

At first glance, explicitly encoding the split offsets would seem to inflate the compression rates. But this is more than compensated by the ability to immediately reconstruct the mesh and use partial information about the connectivity to predict the next symbol.

Labels Our label-based forward decoder with offsets uses C, R, W, S_k, and E. When the length of the current boundary b is three, only labels C and E are possible. They are compressed with a two-entry table that is switched based on the *current degree* of the focus vertex. When b is larger than three, only labels C, R, W, and S are possible. They are compressed with a four-entry table that is switched based on a combination of the current focus degree and the previous label. Labels of type C that immediately follow a label of type E do not need to be compressed explicitly. This happens once for each split operation S: each such operation pushes a boundary onto the stack and the first operation after this boundary is popped from the stack is of type C. The offset value k associated with every label of type S can be compressed using $\log_2(b)$ bits where b is the current length of the boundary.

Degrees Our degree-based forward decoder with offsets uses V_k and $S_{k,l,m}$. It maintains a slot count with each boundary vertex. Should the slot count at the focus drop to zero it performs a right operation. Should the slot count at the focus^{-2} (that is, the second vertex counting clockwise from the focus) drop to zero is performs a wart operation. Otherwise it checks the slot count at the pre-focus: if it is less than three the next operation must be an "add." Else it explicitly decodes whether the next operation is an "add" or a "split" using a binary arithmetic context. The decoder switches between different contexts based on the degree of the pre-focus as split operations are more likely to involve vertices of higher degree.

In case of an "add" operation the corresponding vertex degree is decoded. Should, however, the slot count of each of the b boundary vertices be one, then the new vertex degree must equal b and can be omitted. This happens every time a boundary ends and is illustrated in Figure 3(b). Hence, the last vertex degree plus one additional vertex degree for every split operation can always be omitted.

The following heuristic takes this trick one step further: A slot count of one at the focus, the pre-focus, or their immediate neighbors gives us an additional constraint on the encoded vertex degree. This is illustrated in Figure 3(a) where we have three such *one-slots*. They imply that the new vertex must have a degree of six or more since it must be incident to (a) the four unprocessed triangles along the boundary and (b) to at least two more triangles for there not to be a chord. Eliminating degrees that are not possible from our context tables during arithmetic coding will often lower the total coding costs below the entropy of the vertex degree distribution.

In case of a split operation, the offset k, which represents a distance in vertices along the boundary, can be specified with $\log_2(b)$ bits where b is the current length of this boundary. The counts l and m, which indicate how many slots move over to the split-off boundary part at the two vertices shared by both boundary parts, can be specified with $\log_2(s-2)$ bits where s is the current slot count at the respective vertex. Further optimization for coding the offset k is possible by taking into account that it references a vertex with a slot count of two or higher. Skipping boundary vertices that only have a slot count of one typically gives us a smaller offset k_2 that can then be specified with $\log_2(b_2)$ bits where b_2 is the current number of vertices with a slot count of two or higher on the boundary.

4.2 Forward without Offsets

For offset-less forward decoding we could try to adapt either of the two approaches [11, 12] proposed for Edgebreaker's CLERS sequence. Like the original decoder [11], we could perform two passes over the sequence. The first pass would pre-compute all split offsets by adding up changes in boundary length induced by the symbols between corresponding pairs of S and E. These offsets would then be used in a second pass over the symbols for decoding with offsets as described in the previous section.

For our label sequence this offset computation could be done by maintaining a running total to which we add and subtract a value based on the label type. However, there is no obvious corresponding approach for our degree sequence. It is not possible to simply precompute these offsets by adding up some value for every vertex degree as the order in which they appear makes a difference too.

Alternatively, like the Wrap&Zip decoder [12], we could perform an initial pass that constructs a triangle spanning tree and gives each unmatched edge a "zip" orientation. Then, in a subsequent pass, we would identify pairs of unmatched edges based on their "zip" direction. Adapting the Wrap&Zip technique to our label sequence seems straightforward, but there is again no obvious corresponding method for also doing this with the degree sequence.

In order to emphasize the duality of the two symbol sequences we propose a new two-pass decoding scheme that works similarly for both the label and the degree sequence. In the first pass it constructs a vertex spanning tree that has specially marked leaf nodes attached. In the second pass it reconstructs the triangles by extending these leaves to attach to a vertex of the spanning tree.

We say that our decoding algorithm constructs a *leaf-tree* from which the triangulation can be reconstructed through a *closure* operation. A *leaf-tree* is a rooted planar tree of n nodes with $2n-5$ leaf nodes that are attached in a particular manner. Any vertex spanning tree of a triangulation can be converted into a leaf-tree by walking around the vertex spanning tree and attaching a leaf whenever the walk crosses for the first time an edge that is not part of the spanning tree (a similar tree can be constructed by attaching a leaf on the second crossing [7]). The *closure* operation repeatingly extends *eligible leaves* into edges by attaching them to a vertex so that a triangle is formed. Eligible leaves are those that are directly followed (in walk direction) by two consecutive edges that are either part of the spanning tree or have already been formed. The vertices that eligible leaves are extended to are the ones at the other end of these two consecutive edges. By keeping track of eligible leaves the closure operation can reconstruct the triangulation in linear time.

The concept of a leaf-tree is inspired by Turan's early work on coding of planar graphs [15]. His algorithm walks around a spanning tree and attaches two leaves of different type for each of the two crossings of a non-spanning tree edge. For the first crossing the leaves are of type *opening bracket* and for the second crossing they are of type *closing bracket*. The original planar graph can be reconstructed by matching up corresponding pairs of brackets. For fully triangulated graphs we may drop either the opening or the closing brackets, which gives us a leaf-tree [7]. The leaves of the leaf-tree we use for decoding correspond to Turan's opening brackets.

The planar *two-tree* that is used in Poulalhon and Schaeffer's work on optimal coding of triangulations [9] is also a leaf-tree, albeit a special kind. Poulalhon and Schaeffer construct these two-trees by walking around a very particular vertex spanning tree for which the walk crosses at each node exactly two non-spanning tree edges for the first (or for the second) time. Hence, their two-trees are special leaf-trees for which (nearly) each node has two leaves.

Both the label sequence and the degree sequence describe the construction of a leaf-tree that can be followed by a closure operation to reconstruct the triangulation. For the degree sequence we do not construct a leaf-tree with all leaves fully grown (i.e. it's not in full bloom), but create a leaf tree that grows leaves as the closure progresses by storing a *leaf growth potential* with each node.

The compression rates of these two-pass decoders (see Table 1) are comparatively poor. This is due to the lack of a partially reconstructed mesh when decompressing the symbol sequence. There is no partially connectivity as context for predicting the next symbol.

Labels Our label-based forward decoder without offsets uses C, R, W, S, and E. The decoder initializes the leaf-tree as two nodes connected by an edge and starts reading the labels. When it reads a C it adds a new node to the tree and a leaf on its left. When it reads an R it adds a leaf on the left. When it reads a W it adds a leaf on the right. When it reads an S it adds a branching point and a leaf on the right. When it reads an E it jumps to the most recent branching point or terminates if there are none left. The label sequence is compressed using an order-4 adaptive arithmetic coder.

Degrees Our degree-based forward decoder without offsets uses V_d, W, S, and E. The decoder initializes the leaf-tree as two nodes connected by an edge. When it reads a V_d it adds a new node to the tree, sets its *leaf growth potential* to *d-2* and adds a leaf on the left. When it reads a W it adds a leaf on the right and decrements the leaf growth potential by one. When it reads an S it adds a branching point and a leaf on the right and decrements the leaf growth potential by two. When it reads an E it jumps to the last branching point or terminates if there are none left. The degree sequence is compressed using an order-2 adaptive arithmetic coder.

4.3 Reverse

For decoding from the reverse sequences we perform a *reversed* replay of the encoding process. For both the label and the degree sequence our decoder is essentially an adapted Spirale Reversi decoder [6]. We use ideas from Szymczak [13] to improve compression of the symbols. The reverse decoders consistently achieve the best compression rates. This is in agreement with Szymczak's results [13] that reverse decompression of Edgebreaker labels outperforms both the TG coder [14] and its variation by Alliez and Desbrun [1], that are both bound by the vertex degree entropy.

Labels Our label-based reverse decoder uses C, R, W, S, and E. We decompress the next label using a context that depends on the last label and the current degree of the focus. We distinguish only degrees between 2 to 11 and clamp higher degrees to 11. Since the first operation after popping a boundary from the stack must be a C operation the corresponding label is not explicitly encoded. Similarly, the first operation following an E operation must be an R operation and the corresponding label can be omitted.

Degrees A degree-based reverse decoding uses V_k, W, S, and E. We decompress the next operation using a context that depends on the last label and the current degree of the focus. We distinguish only degrees between 2 to 11 and clamp higher degrees to 11.

meshes		operations		without offsets		with offsets		reverse		degree	TG coder	
name	vertices	warts	splits	labels	degrees	labels	degrees	labels	degrees	entropy	before	after
horse	48,485	1,976	137	1.63	1.61	1.47	1.38	1.37	1.36	1.42	1.43	1.37
dinosaur	56,194	3,549	310	1.90	1.98	1.76	1.66	1.63	1.62	1.65	1.66	1.63
rabbit	67,039	2,860	142	1.71	1.74	1.58	1.51	1.49	1.48	1.55	1.55	1.50
igea	134,345	4,461	163	1.58	1.51	1.40	1.34	1.32	1.31	1.39	1.40	1.33
armadillo	172,974	10,101	664	1.84	1.97	1.79	1.72	1.67	1.65	1.74	1.74	1.70
isis	187,644	5,540	218	1.55	1.54	1.42	1.36	1.34	1.33	1.44	1.44	1.35

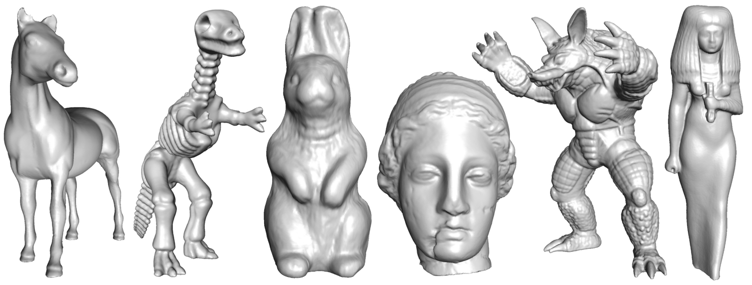


Table 1: For each mesh we list the number of vertices and the number of wart and split operations performed by the the early-split coder. We report compression rates in bits per vertex (bpv) that are achieved by the six possible label or degree sequences. For comparison we also give the entropy of the vertex degree distribution. The last column lists the performance of the original TG coder [14] before and after integrating our novel heuristic for improving the compression of vertex degrees.

5 Compression Results

Our implementation of the early-split coder produces a supersequence containing the labels, degrees, and offsets for all six decoding methods. We have implemented six compressors that take this sequence as input but compress only those symbols used by their respective decompression schemes. The schemes that operate on forward sequences with offsets and on reverse sequences make use of a partially reconstructed mesh for context-based compression of the symbols. To create the same context that is available at decompression time, the compressor simulates the decoding process.

We have run experiments on triangle meshes with vertex counts between 50 and 200 K. We limit our test set to single-component meshes of sphere topology to save us from implementing the treatment of holes and handles for six different coders. This can be done in various ways but has little impact on the total compression rate.

In Table 1 we report for each mesh its total vertex count and the number of wart and split operation that occur during early-split coding. Side by side we list the compression rates for all six encodings in bits per vertex (bpv). These bit-rates are achieved using all the optimizations described in the Section 4. For comparison we also report the entropy of the vertex degree distribution.

Forward decoding without offsets gives the worst compression rates. Decoding is done in two passes so that is not possible to predict the next symbol from a partially reconstructed mesh. Reverse decoding gives the best bit-rates but requires an additional pass to reverse the symbols. This can be disadvantageous when compressing large meshes [4]. For forward decoding with offsets bit-rates are slightly better when using degrees instead of labels because of our new heuristic can only be used in degree-based mode.

6 Conclusion

The main goal of this work was to illustrate the duality of label-based and degree-based coding, *not* to propose yet another coding scheme. For this we designed a connectivity coder that can either produce labels and operate like Edgebreaker and the Cut-border Machine or produce degrees and operate like the TG coder. In either mode it traverses vertices and triangles in the exact same order allowing a side-by-side comparison of label and degree coding.

One insight of this unified view is that for compression it is more important which decoding method we use and not whether we code with labels or degrees. The best rates are achieved by reverse decoding, whereas forward decoding without offsets gives the worst results. Only for forward decoding with offsets the bit-rates suggest that degree coding is a better choice than label coding. This is due to a novel heuristic that makes use of the slot information along the boundaries, which is maintained only by a degree-based replay decoder. We can use same heuristic to bring the compression rates of the original TG coder below the entropy of the vertex degree sequence. This had previously not even been possible with the more complex adaptive-conquest heuristic of Alliez and Desbrun [1].

The other insight is how to do degree coding without storing explicit split offsets. We have previously shown that simply adding "end" symbols is not sufficient to make split offsets implicit to the symbol sequences produced by a degree coder [8]. With early-split coding the new "wart" symbol is easily identified as the additional information that is necessary. For the first time we can present degree-based decoders—one operating in two passes, the other in reverse—that, just like Edgebreaker, do not need explicit offsets.

Acknowledgements

The horse model is courtesy of Greg Turk's Large Model Archive at Georgia Tech. The dinosaur, rabbit, igea, and isis models are courtesy of Cyberware. The armadillo model is courtesy of Stanford's 3D Scanning Repository.

References

[1] P. Alliez and M. Desbrun. Valence-driven connectivity encoding for 3D meshes. In *Eurographics'01 Proceedings*, pages 480–489, 2001.

[2] S. Gumhold. Improved cut-border machine for triangle mesh compression. In *Erlangen Workshop on Vision, Modeling and Vis.*, 1999.

[3] S. Gumhold and W. Strasser. Real time compression of triangle mesh connectivity. In *SIGGRAPH'98 Proceedings*, pages 133–140, 1998.

[4] M. Isenburg and S. Gumhold. Out-of-core compression for gigantic polygon meshes. In *SIGGRAPH'03 Proc.*, pages 935–942, 2003.

[5] M. Isenburg and P. Lindstrom. Streaming meshes. In *Visualization'05 Conference Proceedings*, pages 231–238, 2005.

[6] M. Isenburg and J. Snoeyink. Spirale reversi: Reverse decoding of the Edgebreaker encoding. In *Proceedings of 12th Canadian Conference on Computational Geometry*, pages 247–256, 2000.

[7] M. Isenburg and J. Snoeyink. Graph coding and connectivity compression. *draft:* http://www.cs.unc.edu/˜isenburg/research/.

[8] M. Isenburg and J. Snoeyink. On the non-redundancy of split offsets in degree coding. *draft:* http://www.cs.unc.edu/˜isenburg/research/.

[9] D. Poulalhon and G. Schaeffer. Optimal coding and sampling of triangulations. In *30th International Colloquium on Automata, Languages and Programming (ICAZLP)*, pages 1080–1094, 2003.

[10] J. Rossignac. Just-in-time upgrades for triangle meshes. In *3D Geometry Compression, Course 21, SIGGRAPH'98*, pages 18–24, 1998.

[11] J. Rossignac. Edgebreaker: Connectivity compression for triangle meshes. *IEEE Trans. on Vis. and Computer Graph.*, 5(1):47–61, 1999.

[12] J. Rossignac and A. Szymczak. Wrap&zip: Linear decoding of planar triangle graphs. *The Journal of Comp. Geom., Theory and Appl.*, 1999.

[13] A. Szymczak. Optimized edgebreaker encoding for large and regular meshes. In *Data Compression Conference'02*, page 472, 2002.

[14] C. Touma and C. Gotsman. Triangle mesh compression. In *Graphics Interface'98 Conference Proceedings*, pages 26–34, 1998.

[15] G. Turan. Succinct representations of graphs. *Discrete Applied Mathematics*, 8:289–294, 1984.

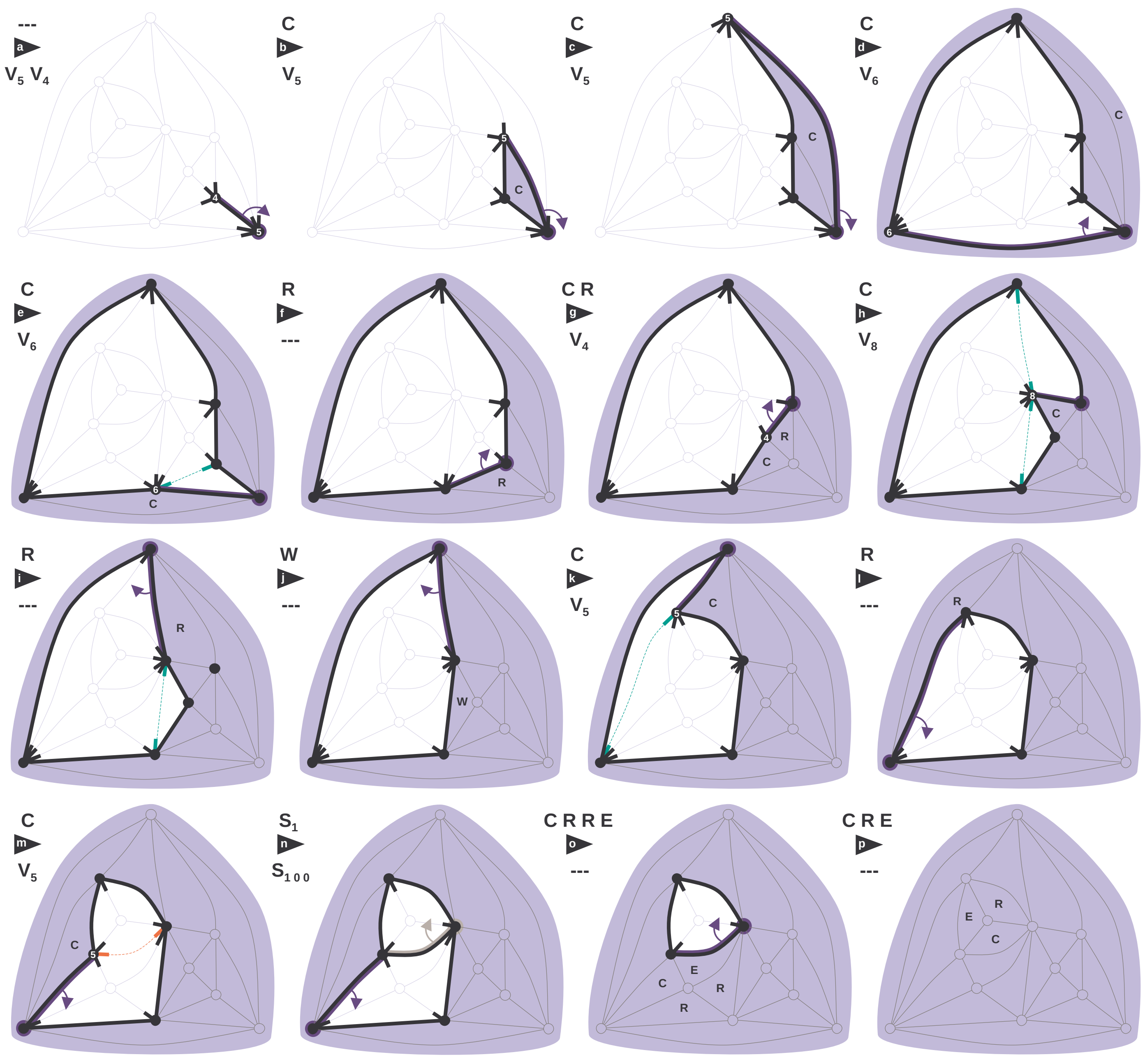


Figure 5: An example of **replay** decoding from a forward label or degree sequence with offsets. The label-based replay reads symbols C, R, W, S_k, and E (shown above the arrow). The degree-based replay reads symbols V_d and $S_{k,i,j}$ (shown below the arrow). The slots on the boundary are only meaningful for degree-based decoding. Because decoding is an *exact* replay of encoding this Figure also illustrates the encoding process.

Label-based: (a) The decoder creates the initial boundary using two new vertices. **(b)** The decoder performs a C operation: it forms a triangle at the gate using a new vertex. **(c–e)** Three more C operations. **(f)** The decoder performs an R operation: it forms a triangle by connecting focus^{-1} with focus^{+1}, which becomes the new focus. **(g–i)** More C and R operations. **(j)** The decoder performs a W operation: it forms a triangle by connecting focus^{-1} with focus^{-3}. **(k–m)** More C and R operations. **(n)** The decoder performs an S_k operation: it creates an edge by connecting focus^{-1} with focus$^{-(k+3)}$, which becomes the focus of the split-off boundary loop that is temporarily pushed onto a stack. **(o)** After performing operations C, R, and R, the decoder performs an E operation. This ends the current boundary loop and the decoder pops a boundary off the stack. **(p)** After performing operations C, R, and E, the decoder terminates because the stack is empty.

Degree-based: (a) The decoder reads the first two vertex degrees V_5 and V_4 and sets the slot count of the initial boundary. **(b)** The decoder reads vertex degree V_5, performs a C operation, and updates the slots. **(c–e)** The decoder reads vertex degrees V_5, V_6, and V_6, performs C operations, and updates slots. **(f)** The decoder performs an implicit R operation because the slot count at the focus has dropped to zero. **(g–i)** Vertex degrees V_4 and V_6 are processed. **(j)** The decoder performs an implicit W operation because the slot count of focus^{-2} reaches zero. **(k–m)** Vertex degrees V_5 and V_5 are processed. **(n)** The decoder reads $S_{k,i,j}$ and performs an S_k operation. The numbers i and j specify that $i{+}1$ of the slots at focus^{-1} and $j{+}1$ of the slots at focus$^{-(k+3)}$ should move over to the corresponding vertices on the split-off boundary loop. **(o)** The decoder implicitly performs C, R, R, and E because all four boundary vertices have a slot count of one. **(p)** The decoder implicitly performs C, R, and E because all three boundary vertices have a slot count of one.

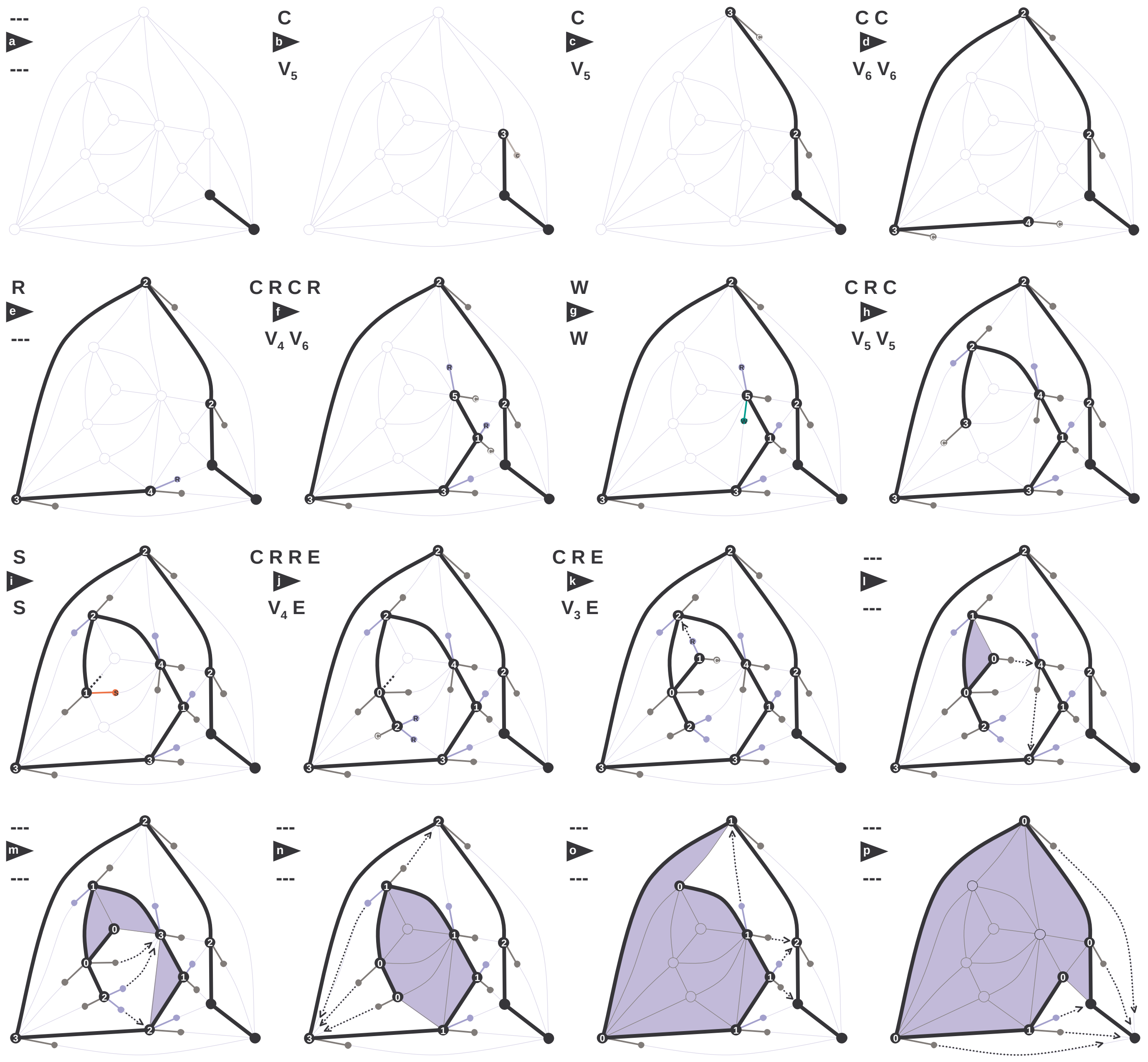


Figure 6: An example of **leaftree** decoding from a forward label or degree sequence without offsets. Decoding happens in two passes: first the construction of the full leaftree, then the closure of the leaftree. The label-based decoder reads symbols C, R, W, S, and E. The degree-based decoder reads symbols V_d, W, S, and E. The light-blue leaves are only meaningful for label-based decoding, whereas the numbers on the vertices are only meaningful for degree-based decoding.

Label-based: **(a)** The decoder creates the initial leaftree using two new vertices. **(b)** The decoder reads a C: it uses a new vertex as the next node and attaches a left leaf. **(c–d)** More C. **(e)** The decoder reads an R: it attaches another left leaf to the current node. **(f)** More C and R. **(g)** The decoder reads a W: it attaches a right leaf to the current node. **(h)** More C and R. **(i)** The decoder reads an S: it attaches a right branch and a right leaf to the current node. **(j)** More C and R, then the decoder reads an E: it continues at the last branch. **(k)** More C, R, and then an E: it terminates since there are no other branches. **(l)** Leaftree closure: a clockwise leaf-edge-edge sequence is closed into a triangle. **(m)** Two more closures. **(n)** Three more. **(o)** Four more. **(p)** Another four and the next five complete the closure.

Degree-based: **(a)** The decoder creates the initial leaftree using two new vertices. **(b)** The decoder reads a V_5: it uses a new vertex as the next node, attaches a left leaf, and sets the potential leaf count to 3. **(c)** Another V_5. In addition, the potential leaf count of the parent node is decremented. **(d)** Two V_6s. **(e)** Nothing happens (remember: the degree-based encoding does not contain R symbols and the light-blue leaves are only meaningful for label-based decoding). **(f)** More V_ks. **(g)** The decoder reads a W: it attaches a right leaf to the current node. **(h)** Two more V_5s. **(i)** The decoder reads an S: it attaches a right branch and a right leaf to the current node. **(j)** After reading an E the decoder continues at the most recently attached branch. **(k)** After reading another E the decoder terminates. **(l–p)** Leaftree closure: nodes without children or whose children are all without leaves turn their remaining leaf growth potential into left leaves. Intuitively speaking, the degree-based decoder grows all those blue leaves that it was not explicitly told about just in the moment they are needed for forming a triangle.

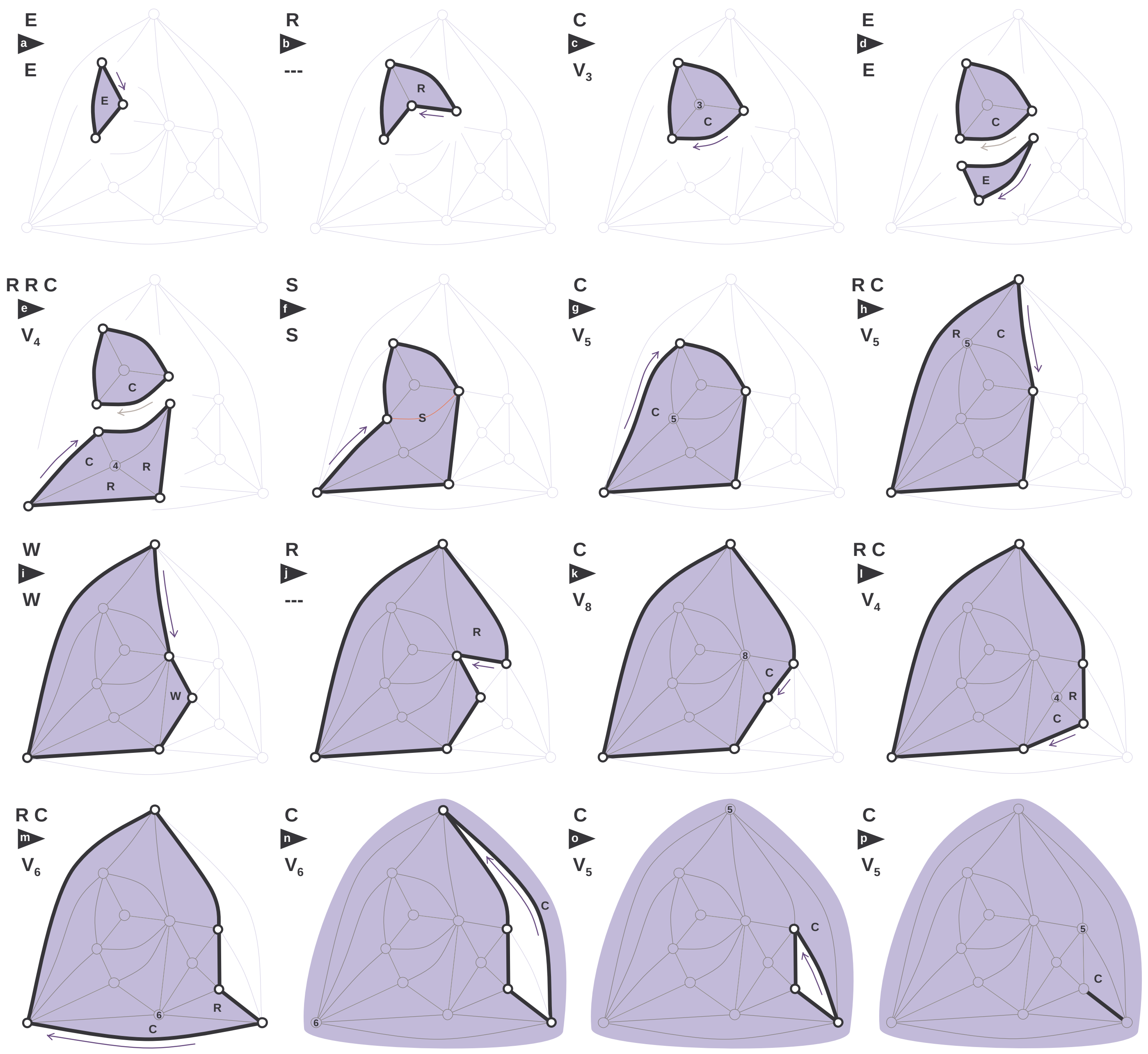


Figure 7: An example of **reverse** decoding from the reversed label or degree sequence without offsets. The label-based decoder reads symbols C, R, W, S, and E. The degree-based decoder reads symbols V_d, W, S, and E. The reverse decoders perform the boundary updates that happen during encoding in the exact reverse order:

Label-based: **(a)** The decoder reads an E: it creates the initial boundary in form of a single triangle. **(b)** The decoder reads label R: it creates a triangle at the gate. **(c)** The decoder reads label C: it creates a triangle that completes the focus. **(d)** The decoder reads another label E: it pushes the current boundary on the stack and creates a new boundary in form of a single triangle. **(e)** The decoder reads labels R, R, and C and creates triangles accordingly. **(f)** The decoder reads label S: it pops a boundary from the stack and merges it with the current boundary. **(g–h)** More labels R and C are read and processed. **(i)** The decoder reads label W: it creates a triangle at the edge following the gate. **(j–o)** More labels R and C are read and processed. **(p)** After processing label C the decoder terminates because the boundary has length two.

Degree-based: **(a)** The decoder reads an E: it creates a triangle. **(b)** Nothing happens. **(c)** The decoder reads vertex degree V_3: it creates as many triangles as needed (here: two) so that the focus has a degree of three. **(d)** The decoder reads another E: it pushes the current boundary on the stack and creates a new boundary in form of a single triangle. **(e)** The decoder reads vertex V_4: it creates three triangles so that the focus has a degree of four. **(f)** The decoder reads an S: it pops a boundary from the stack and merges it with the current boundary. **(g)** The decoder reads an V_5: it creates as many triangles as needed (here: one) so that the focus has a degree of five. **(h)** Another V_5. **(i)** The decoder reads a W: it creates a triangle at the edge following the gate. **(j)** Nothing happens. **(k–o)** More vertex degrees are read and triangles are created accordingly. **(p)** After processing this V_5 the decoder terminates because the boundary has length two.