Efficient Traversal of Mesh Edges using Adjacency Primitives

Pedro V. Sander
Hong Kong UST

Diego Nehab
Microsoft Research

Eden Chlamtac
Princeton University

Hugues Hoppe
Microsoft Research

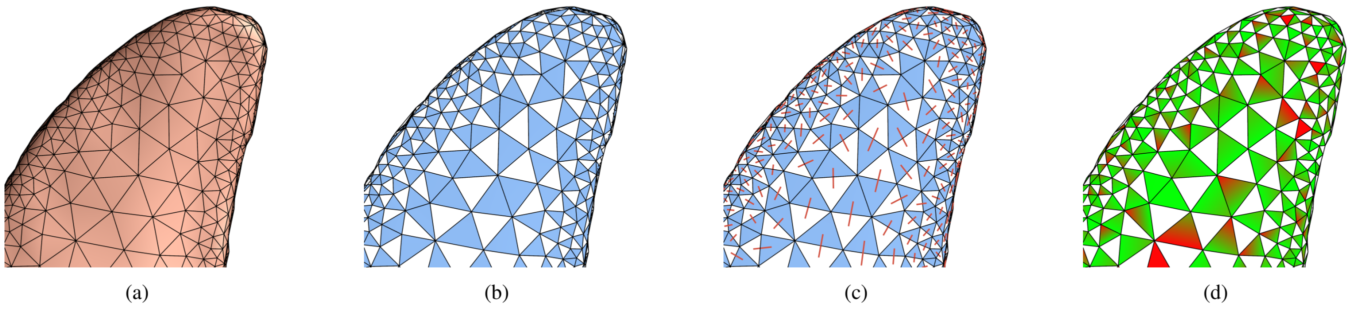


Figure 1: (a) Our goal is to enable efficient processing of edges in a triangle mesh using adjacency primitives. (b) We select a minimal subset of triangles (blue) that covers all mesh edges. (c) Each remaining triangle (white) is assigned to a cover triangle (indicated by red segments). (d) We encode each triangle pair as a triangle-with-adjacency primitive, and order these primitives for vertex cache locality (cache hits in green, misses in red). This new representation reduces storage, bandwidth, and GPU computation, resulting in substantial gains for a variety of edge-processing techniques.

Abstract

Processing of mesh edges lies at the core of many advanced real-time rendering techniques, ranging from shadow and silhouette computations, to motion blur and fur rendering. We present a scheme for efficient traversal of mesh edges that builds on the adjacency primitives and programmable geometry shaders introduced in recent graphics hardware. Our scheme aims to minimize the number of primitives while maximizing SIMD parallelism. These objectives reduce to a set of discrete optimization problems on the dual graph of the mesh, and we develop practical solutions to these graph problems. In addition, we extend two existing vertex cache optimization algorithms to produce cache-efficient traversal orderings for adjacency primitives. We demonstrate significant runtime speedups for several practical real-time rendering algorithms.

Keywords: real-time rendering, silhouettes, shadow volumes, vertex locality, programmable geometry shader.

1 Introduction

Triangles are a widespread rasterization primitive, so there is a large body of work on optimizing the traversal of triangle meshes for efficient rendering, e.g. using triangle strips [Evans et al. 1996; Xiang et al. 1999; Estkowski et al. 2002], managed vertex buffers [Deering 1995; Chow 1997], and indexed strips with vertex caching [Hoppe 1999; Lin and Yu 2006; Sander et al. 2007; Chhugani and Kumar 2007]. In particular, current GPUs include a vertex cache to allow reuse of post-shaded vertices among several adjacent triangles, thereby achieving significant reduction in both memory bandwidth and vertex shader computation.

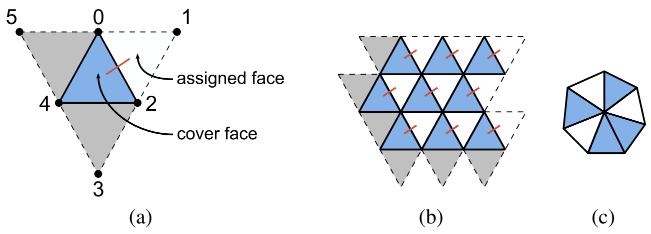


Figure 2: (a) Our mesh traversal primitive is the triangle-with-adjacency, which we use to process up to 3 edges and 2 faces. (b) It provides an optimal cover in regular mesh regions. (c) On the other hand, around any odd-degree vertex, at least one adjacent edge must be covered twice.

Many advanced rendering techniques process not only mesh faces but also mesh edges. Applications include shadow volumes [Crow 1977], silhouette rendering [Hertzmann 1999; Gooch and Gooch 2001], motion blur [Wloka and Zeleznik 1996], fur rendering [Lengyel et al. 2001], and wireframe rendering. In most techniques, processing an edge requires access to its two adjacent faces. This has motivated the introduction of *adjacency primitives* in the latest graphics systems [Blythe 2006]. The primitives are processed in a new programmable unit of the graphics pipeline, the *geometry shader*, which reads a primitive, performs computation, and emits a variable number of new primitives.

In this paper, we present a general scheme that optimizes the traversal of a mesh for efficient GPU processing of its edges. To our knowledge, this is the first attempt at such an optimization. Some graphics applications require traversal of triangles in addition to edges, and we explore how this can be achieved efficiently in the same traversal pass, or in separate passes using the same vertex and index buffers.

Approach Our basic strategy is to traverse the mesh using a list of triangle-with-adjacency primitives. As shown in Figure 2a, each such primitive uses 6 indices to encode a central *cover face* as well as its 3 adjacent faces. In principle, the primitive permits the processing of the cover face itself, all its 3 edges, and all its 3 adjacent faces. Our goal is to process all mesh edges and triangles *exactly once*, using the same list of primitives, while keeping the overall number of primitives to a minimum.

Because the geometry shaders consider many primitives simultaneously with SIMD parallelism on the GPU, it is crucial for computational efficiency that processing each primitive requires the same sequence of steps. To maximize this SIMD efficiency, we allow each adjacency primitive to process 1 or 2 faces at a time (Figure 2a): the central cover face, and an optional adjacent face that we refer to as the *assigned face*. To further improve SIMD parallelism, we place some restrictions on the selection of assigned faces, as discussed in Section 5.

Note that, in regular mesh regions, all edges and faces can be covered by introducing primitives for only half of the triangles, i.e. all the faces "pointing in the same direction" (Figure 2b), each one assigned to a face pointing in the opposite direction. On the other hand, around any irregular vertex of odd degree, at least one edge must be covered twice, and therefore some redundancy is inevitable (Figure 2c). Nevertheless, because most triangles-with-adjacency primitives encode two faces, the index buffer contains only about 3 indices per mesh triangle (one primitive for every 2 mesh triangles, 6 indices per primitive), and therefore has approximately the same memory cost as the widely used indexed triangle list representation.

Algorithm overview The problems we face can be formulated as graph theoretical problems on the dual graph induced by the triangle mesh. In particular, we show that selecting a minimum number of primitives to cover all edges reduces to a minimum vertex cover problem, and that assigning the remaining faces to these primitives reduces to two bipartite matching problems. Our algorithm has three major steps, as shown in Figure 1:

1. The vertex **cover** problem is NP-complete, but fortunately in our particular setting we can rely on a fast approximation using a stochastic algorithm. Moreover, by deriving a good lower bound on the number of cover faces, we are able to show that, for practical triangle meshes, the approximate solutions are within a few percent of optimal;
2. The face **assignment** problem reduces to two bipartite matching problems, and can therefore be solved quickly in $O(n^{1.5})$ time; e.g. only 4 sec for a mesh with $n = 75{,}000$ vertices. In particular, we prove the surprising result that perfect matchings are guaranteed to exist, even with the added restrictions introduced for SIMD efficiency;
3. Having determined a set of triangles-with-adjacency, we optimize their **ordering** to maximize vertex cache reuse.

The cover and assignment problems are always solved in an offline preprocess. Because ordering efficiency depends on cache size, we provide two ordering algorithms: a slower, careful scheme that assumes prior knowledge of the cache size, and a faster scheme that can be run at load time based on the specific hardware.

Contributions:

- A construction that minimizes the number of adjacency primitives needed for the processing of mesh *edges* (Section 4);
- An accurate lower bound algorithm that confirms the quality of this minimization (Appendix);
- A construction that allows this reduced set of primitives to process all mesh *faces*, retaining SIMD efficiency (Section 5);
- The first modified algorithms that perform vertex cache optimization on lists of triangles-with-adjacency (Section 6).

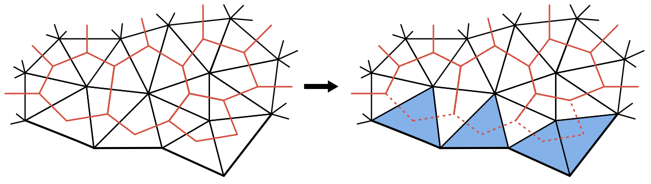


Figure 3: (a) Primal graph associated to a mesh (black), and its dual (red). (b) Automatic inclusion of boundary faces into cover.

2 Previous work

Optimizing the traversal of mesh edges on the GPU is a relatively unexplored area. One prior technique for computing shadow volumes entirely on the GPU is to introduce a degenerate quadrilateral on each edge of the input mesh, and to selectively translate vertices to expand a fraction of these degenerate faces to form the sides of the shadow volume [Brennan 2002]. However, the technique requires splitting the mesh vertices in a preprocess, resulting in a mesh that is 6 times as large as the original.

Some related techniques for GPU rendering of silhouettes or fins also introduce degenerate quadrilaterals on edges [Card and Mitchell 2002; McGuire and Hughes 2004], and therefore also require memory buffers with many additional vertices and faces.

Several sample programs demonstrate applications of the new triangle-with-adjacency primitive [Microsoft Corp 2007; Tariq 2007]. The basic approach in all these programs is to instantiate a primitive for each input triangle. To avoid processing interior edges twice, a test is performed in the geometry shader to appropriately skip half of the edges. This test is based either on the triangle orientation at silhouettes, or on per-edge ordering of vertex indices. Note that such local branching leads to inefficient SIMD processing. In contrast, our scheme generates roughly half as many primitives, and moreover most of these primitives process all 3 of their adjacent edges, resulting in excellent SIMD utilization.

3 Notation

We describe our processing pipeline in the next three sections using the following notation. A triangle mesh $M(V, E, T)$ is defined by a set of vertices V, edges E, and triangles T. Each triangle is formed by three vertices $\{v_i, v_j, v_k\}$ in V, and defines three edges: $\{v_i, v_j\}$, $\{v_j, v_k\}$, and $\{v_k, v_i\}$. The set E contains all edges defined by the triangles in T.

As shown in Figure 3a, two undirected graphs can be associated with mesh M. The *primal* graph $G(V, E)$ has the same vertices V and edges E. The *dual* graph $G'(V', E')$, on the other hand, has one vertex in V' for each triangle in T. The edges in E' connect two vertices in V' if and only if the associated triangles in T share an edge in E.

4 Covering all edges

Recall that our goal is to create a minimum set of triangle-with-adjacency primitives that allow the processing of all edges in a mesh $M(V, E, T)$. This is equivalent to finding the minimum vertex cover $C \subset V'$ in the *dual* graph $G'(V', E')$. In general, a set of vertices $B' \subset V'$ may be adjacent to the mesh boundary, and must therefore be part of the cover. For meshes with boundaries, we find the minimal vertex cover C^* on the reduced graph $G^*(V' \setminus B', E^*)$ (i.e., G' with all boundary vertices removed, as shown in Figure 3b), and return $C = C^* \cup B'$.

Ideally, we would like to find the *minimum* vertex cover in G^*. However, the problem is NP-complete [Garey and Johnson 1977], even for the graphs with maximum degree 3 arising from manifold triangle meshes. Therefore, we look for heuristics that produce

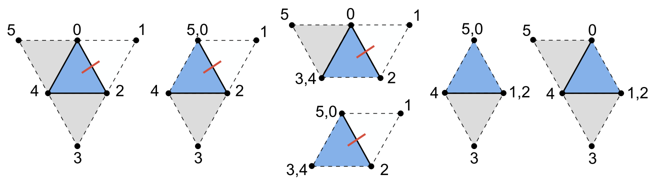


Figure 4: For fast geometry shader evaluation, we allow only 6 configurations of the triangle-with-adjacency primitive. Each primitive processes the central triangle (blue), a subset of its edges (solid lines), and an optional matched triangle (white). Repeated vertex indices identify edges that should not be processed, a missing adjacent face, or both.

good *minimal* covers, in which no vertex can be trivially removed without leaving an uncovered edge. Note that the cover C is also minimal, since every $v' \in B'$ is needed to cover a corresponding boundary edge, and by minimality of C^*, no $v' \in C^*$ can be removed without exposing an internal edge in G^*.

Several recent stochastic algorithms address the related problems of computing approximations to minimal vertex covers, maximum cliques, or maximum independent sets on large graphs [Grosso et al. 2007; Andrade et al. 2008]. Although we could use any of them for our purposes, our implementation is based on the method of Grosso et al. [2007]. This is an excellent maximum clique approximation heuristic, which is related to the minimum vertex cover as follows: C is a minimum vertex cover for graph $G'(V', E')$ if and only if $\overline{C} = V' \setminus C$ is a maximum clique on the complement graph $\overline{G'}(V', \overline{E'})$, where $\overline{E'}$ is the set of all edges not in E'.

The algorithm of Grosso et al. iterates between two steps: a perturbation and a tabu search. Each iteration starts with a maximal clique K, and ends with a new maximal clique K'. First, the perturbation sets $K'{=}K$, adds a random vertex $v \notin K'$, and corrects K' by removing from it all vertices not adjacent to v. Next, the tabu search successively grows K' until it is maximal again, while also attempting to translate it away from K as long as they share at least one vertex. At the end of the iteration, K is replaced by K' if the cardinality of K' is larger.

Fortunately, it is possible to apply the stochastic algorithm to our vertex cover problem without having to explicitly construct the complement set $\overline{E'}$, which would have quadratic complexity. We first modify the algorithm of Grosso et al. by negating all the edge tests, to construct a maximal independent set I. Our minimal vertex cover C is simply the complement $\overline{I}$ of this intermediate result.

We find that the stochastic algorithm produces excellent minimal covers C on a wide variety of input graphs produced from triangle meshes. To quantify the quality of these results, we develop an efficient algorithm that provides a good lower bound on $|C|$. To emphasize that this bound is only used for evaluation purposes, and is not a necessary part of our processing pipeline, we describe it in the appendix. Using this lower bound, we determine that the covers C are typically within 2-3% of optimal (see Table 1). In fact, since the lower bound is not tight, the covers may be closer to optimal than suggested by these numbers.

5 Assigning the remaining triangles

Having created a triangle-with-adjacency primitive centered at each cover face, we assign the uncovered faces to adjacent faces within these primitives. For efficient SIMD load-balancing, we assign at most one uncovered face to each primitive. And to simplify the runtime traversal, we assume that this assigned face is the first adjacent face $\{0, 1, 2\}$ within the primitive (see Figure 4). If the primitive lacks an assigned face, is at the boundary, or if one of its edges has already been covered by a neighboring primitive, we encode a degenerate face by duplicating some vertex indices (also shown in Figure 4). There are two ways to duplicate the index, and we can use them to distinguish between boundary edges and edges that have already been covered.

Note, however, that whenever $\{0, 2\}$ is not a boundary edge and face $\{0, 1, 2\}$ is made degenerate, edge $\{0, 2\}$ also becomes degenerate and cannot be processed by the current primitive. Recall that processing of an edge requires access to both its adjacent triangles, but in this primitive one of them is degenerate. Therefore, the edge must be processed by another primitive: the primitive centered on face $\{0, 1, 2\}$, which must be a cover face too.

Thus, if a non-boundary cover face f_1 lacks an assigned face, it must be adjacent to another cover face f_2, which if it also lacks an assigned face and is not on the boundary, must be adjacent to a third cover face f_3, etc. This chain of restrictions is satisfied if every connected component of cover faces has at least one face that is either adjacent to the mesh boundary or is matched to an uncovered face. This constrains the acceptable matchings, but we will prove that nonetheless it is always possible to find such a perfect *restricted* matching.

Formally, given the minimal cover C in the dual graph G' (including all dual vertices corresponding to boundary faces), we must match each remaining vertex in $\overline{C} = V' \setminus C$ to a vertex in C. Finding an *unrestricted* matching from $\overline{C}$ to C is easy. Let $d(v)$ denote the degree of vertex v. If we consider the bipartite graph $G'' = (\{C, \overline{C}\}, E'(C, \overline{C}))$, where $E'(C, \overline{C}) \subset E'$ is the subset of edges between C and $\overline{C}$, we see that $d_{G''}(v') = 3$ for all $v' \in \overline{C}$ (since v' is not in the cover, all its three neighbors must be), and $d_{G''}(v') \leq 3$ for all $v' \in C$ (a triangle has at most three neighbors). It follows that $\overline{C}$ satisfies the *marriage condition*, which is that for any subset $C' \subseteq \overline{C}$ and its neighbors $\Gamma_{G''}(C') \subseteq C$, we have $|\Gamma_{G''}(C')| \geq |C'|$. Thus Hall's theorem [Hall 1935] guarantees the existence of a matching M from $\overline{C}$ to C.

To express the conditions on the restricted matching, denote by $\{C_i\}$ the partition into connected components of the subgraph of G' induced on C. For each C_i we must ensure that at least one of the following two conditions holds:

1. Some $v' \in C_i$ corresponds to a mesh boundary triangle, or
2. Some $v' \in C_i$ is matched to $\overline{C}$, i.e. it has an assigned face.

The proof of the following lemma (illustrated in Figure 5) describes a construction algorithm for the restricted matching.

Lemma 1. *For any bipartite graph G'' as above, there always exists a perfect restricted matching M' from $\overline{C}$ to C satisfying the above requirements.*

Proof. As we have seen, there always exists some unrestricted perfect matching M from $\overline{C}$ to C (Figure 5a). We must find a matching M' that also covers at least one face in each component C_i that does not contain a boundary face. Let $A \subset \{C_i\}$ denote this set of components. Let us collapse for a moment each $C_i \in A$ to a single vertex, and merge repeated edges. Our problem is to find a perfect matching in this graph from A to $\overline{C}$. We again use Hall's theorem, which we can apply if the marriage condition is satisfied. That is, for any subset $K \subset A$ of non-boundary components, and its neighbors $\Gamma(K) \subseteq \overline{C}$, we must show that $|K| \leq |\Gamma(K)|$. Indeed, let $E'(K)$ be the set of dual edges from K to $\Gamma(K)$ *before merging*. Note that each $C_i \in A$ must have at least three edges in G' going to $\overline{C}$ (since, by minimality of the cover C, the corresponding bounded region in the primal is surrounded by faces in $\overline{C}$). Moreover, every $u' \in \overline{C}$ has degree ≤ 3 (originally every $u' \in \overline{C}$ had degree 3, though some of its edges might not participate in $E'(K)$). Therefore

$$3|K| \leq |E'(K)| \leq 3|\Gamma(K)|.$$

Thus, by Hall's theorem, there exists a matching N from A into $\overline{C}$ (Figure 5b).

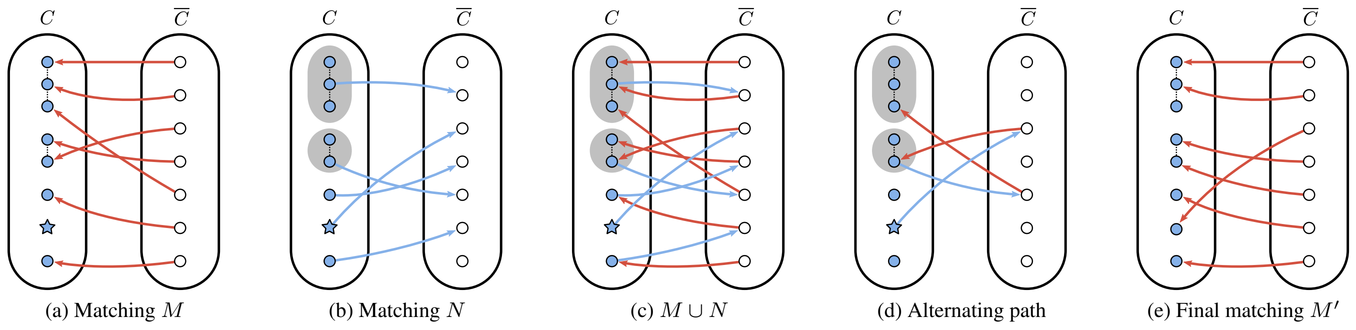


Figure 5: The face assignment corresponds to a restricted matching problem on a bipartite graph (where gray shaded regions denote connected components). (a) The initial unrestricted perfect matching leaves an isolated covered face (star) unmatched. (e) After several steps, our construction is guaranteed to find a final perfect matching such that all connected covered faces have at least one matching.

Now consider the union of the two matchings $M \cup N$ (Figure 5c). These form a disjoint collection of cycles and paths (because every vertex in this graph has degree ≤ 2). Let $v' \in C$ be a vertex participating in matching N but not in M (the star in Figure 5). Then there is a path alternating between C and $\overline{C}$ which starts at v' and ends in a vertex $u' \in C$ which participates in M but not N (Figure 5d). Let P be the set of edges in all such alternating paths (note that they are disjoint). Now take M and flip the membership of all edges in the alternating paths, producing a new perfect matching $M' = (M \setminus P) \cup (P \setminus M)$ (Figure 5e).

Clearly, matching M' also covers all of $\overline{C}$. Let us verify that every $C_i \in A$ participates in this matching. Since A is completely matched by N, every C_i must contain some vertex v' that is matched by N. If it is also matched by M, then it will participate in M' regardless of whether an alternating path going through it is flipped. Otherwise, it is the initial vertex in an alternating path, and thus will participate in M' once this path is flipped. $\square$

To find the bipartite matchings, we use the algorithm by Hopcroft and Karp [1973], as implemented by Rothberg [1985]. The time complexity of this algorithm is $O(|E'|\sqrt{|V'|})$, which is simply $O(n^{1.5})$ on the size of the input mesh.

6 Ordering primitives for vertex locality

The idea of reordering primitives for efficient GPU traversal is not new, but has previously been limited to triangle primitives. Several methods optimize the traversal of indexed triangles in a mesh to maximize the runtime efficiency of a GPU vertex cache. We focus here on the methods that are most related to our approach, and refer the reader to Chhugani and Kumar [2007] for a more complete survey. Hoppe [1999] shows the efficacy of a FIFO cache, and builds successive triangle strips by greedily optimizing their lengths. Lin and Yu [2006] improve caching efficiency by using triangle fans and optimizing their selection based on the age of the vertex in the cache and the number of cache misses that would result. Chhugani and Kumar [2007] further improve the cache efficiency; their approach partitions the mesh triangles into adjacent chains, and then splits the chains into strips using dynamic programming. Sander et al. [2007] develop a fast algorithm that does not require mesh adjacency, has linear complexity independent of cache size, and is therefore suitable for load-time optimization of triangle buffers.

Our contribution is to optimize the traversal of triangle primitives *with adjacency*, created as described in the previous two sections. Because each adjacency primitive involves a larger stencil of up to 6 vertices, naive application of previous reordering methods gives poor results. (We arbitrarily selected [Lin and Yu 2006] as the representative "off-the-shelf" method.)

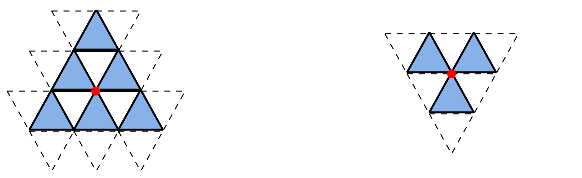

(a) Careful scheme: 6 primitives
(b) Fast scheme: 3 primitives

Figure 6: Each of our reordering algorithms considers a different set of primitives around a central vertex.

We chose to extend two of the prior methods, namely [Lin and Yu 2006] and [Sander et al. 2007], because their strategy of emitting all unvisited primitives adjacent to a central vertex can be generalized to other primitive types. In the results, we denote our extensions of these two methods as the "careful" and "fast" schemes, respectively.

Careful scheme based on extending [Lin and Yu 2006] The algorithm of Lin and Yu proceeds by iteratively selecting a central vertex and emitting all its adjacent unvisited faces. The vertex selection algorithm considers the position of the vertex in the cache, the number of unvisited faces, and the number of cache misses that would result. We modify this cost analysis to consider the ring of *adjacency* primitives that reference the candidate central vertex. In a regular mesh region, there are 6 such primitives, as shown in Figure 6a. We also modify the algorithm to emit the adjacent primitives in the order that minimizes the resulting number of cache misses.

Fast scheme based on extending [Sander et al. 2007] The algorithm of Sander et al. considers fewer candidates for the next central vertex to process, and makes the conservative assumption that unvisited adjacent faces may cause two additional cache misses. With adjacency primitives, this conservative bound must be raised to five cache misses per primitive. Therefore, to obtain good results, we found it necessary to emit only the primitives whose cover face is immediately adjacent to the central vertex; i.e. only up to 3 adjacency primitives in a regular mesh region, as shown in Figure 6b.

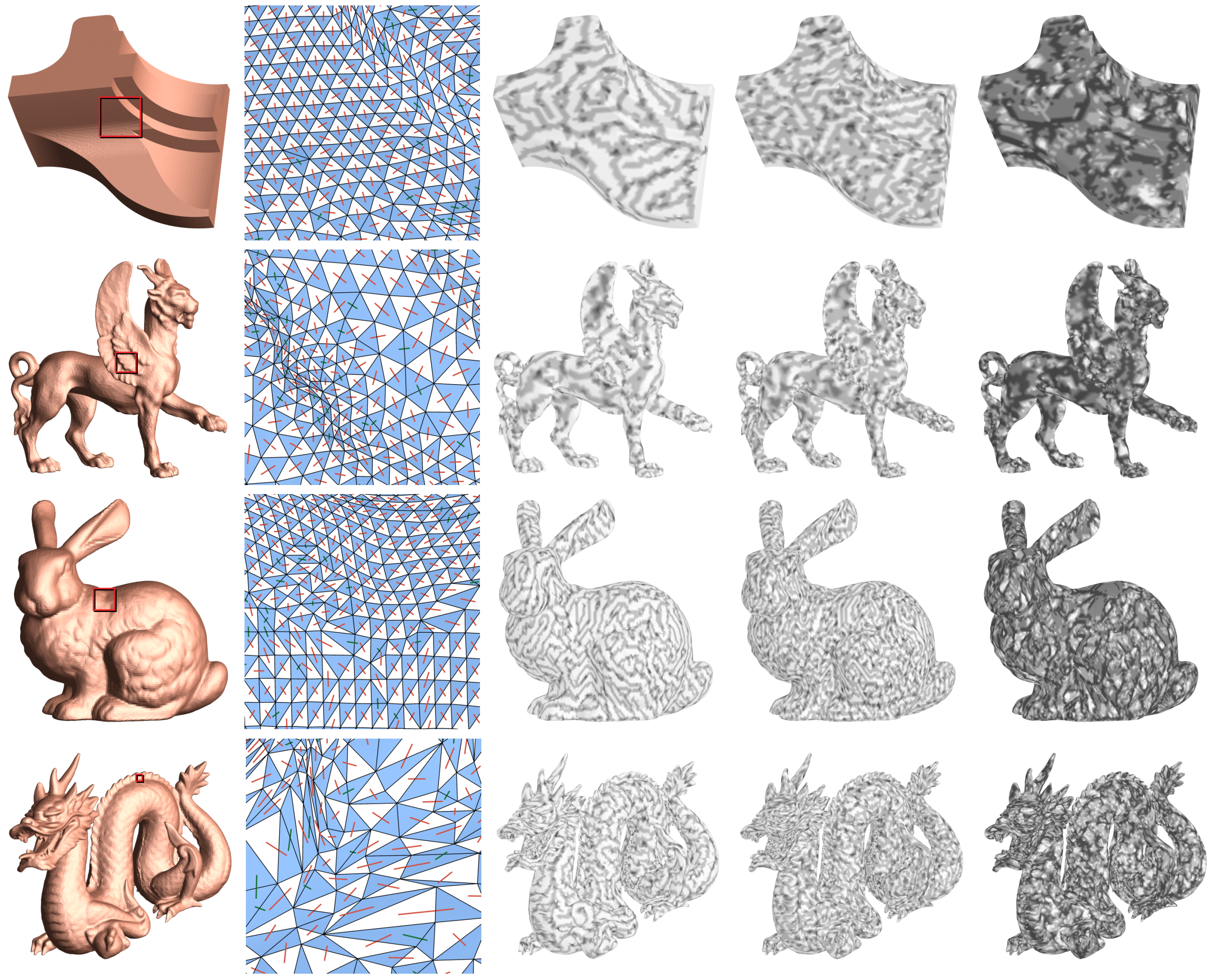
As this scheme takes only a fraction of a second to run even on large meshes (Table 1), it can be used to quickly specialize the primitive ordering to the cache size of a given graphics system. Note that the cover and assignment can still be obtained in a preprocessing stage, since they are unaffected by cache size.

As future work, it would also be interesting to explore extensions of [Hoppe 1999] and [Chhugani and Kumar 2007], which are based on strips.

Table 1: Results of our traversal optimization on several meshes.

							Ordering[†]					
Input mesh		Cover				Assignment	Careful			Fast		
Name	# Faces	Lower bound (B)	Cover (C)	C/B	Time (s)	Time (s)	CM/V*	Gain[‡]	Time (s)	CM/V*	Gain[‡]	Time (s)
fandisk	9,926	5,199 (52.3%)	5,335 (53.7%)	1.026	0.62	0.07	1.496	1.955	1.23	1.755	1.666	0.002
gargoyle	20,000	10,880 (54.4%)	11,225 (56.1%)	1.032	1.42	0.22	1.517	1.794	3.12	1.805	1.508	0.006
feline	41,262	21,937 (53.2%)	22,570 (54.7%)	1.029	3.50	0.70	1.529	1.903	6.17	1.816	1.602	0.014
bunny	69,473	36,382 (52.4%)	37,168 (53.5%)	1.022	8.47	0.92	1.535	1.938	9.33	1.780	1.672	0.023
dragon	150,000	81,520 (54.3%)	84,034 (56.0%)	1.031	50.32	4.02	1.505	1.671	21.73	1.807	1.417	0.053
turtle	267,931	142,390 (53.1%)	146,158 (54.6%)	1.026	55.08	5.95	1.533	1.885	36.38	1.792	1.608	0.099
buddha	1,087,716	591,909 (54.4%)	610,526 (56.1%)	1.031	315.38	341.00	1.501	1.677	212.58	1.798	1.400	1.256

[†]Measurements are based on a cache with 24 entries. *CM/V indicates cache misses per mesh vertex and has a lower bound of 1. [‡]Gain factors indicate reduction in cache misses compared to an off-the-shelf method.



(a) Input mesh (b) Cover faces and assignment (c) Careful vertex cache ordering (d) Fast vertex cache ordering (e) Off-the-shelf vertex cache ordering

Figure 7: Results of our mesh traversal construction on 4 representative meshes from Table 1.

7 Results and applications

Table 1 shows the results of our traversal construction for an assortment of practical meshes. The C/B column represents the ratio between the cover results (C) and our computed lower bound results (B). Remarkably, C/B indicate that Grosso et al. [2007] can find cover solutions that are only 2–3% worse than the lower bound, and therefore at most 2–3% worse than optimal. In practice, both the cover and the assignment steps can be performed efficiently (in seconds for small models, and minutes for large models). Note that this is a pre-processing algorithm and therefore not time-critical. As shown later in the applications, the excellent cover and assignment results translate into significant speedups in practical rendering scenarios.

Figure 9 plots the cache efficiency as a function of cache size. The efficiency, denoted CM/V, is given by the number of cache misses divided by the number of vertices. Lower values are better, and the optimal minimum value is 1, given that each vertex must be processed at least once. Note the substantial reduction in the number of cache misses per vertex for both our careful and fast schemes compared to the ordering produced by a representative prior algorithm designed for triangle primitives without adjacency. As shown in Table 1, the fast scheme is orders of magnitude faster than the careful scheme, and like [Sander et al. 2007], its processing time is independent of the cache size. Figure 7 shows a visualization of the results on 4 meshes from Table 1. The spatial distribution of cache efficiency is conveyed by coloring each mesh vertex according to its total number of cache misses. The color white corresponds to the optimal single cache miss per vertex. Darker shades of gray indicate progressively more cache misses per vertex.

We demonstrate the practical advantages of our efficient edge-processing traversal with three applications (see Figure 8).

Shadow volumes The mesh is rendered as a shadow volume into a screen-space stencil buffer [Heidmann 1991]. The shadow volume is formed within the geometry shader, by displacing triangles to form a front cap and a rear cap (depending on whether they face the light), and by extruding silhouette edges (with respect to the light) to form quadrilaterals spanning these caps. As a baseline, we used sample program ShadowVolume10 of the DirectX 10 SDK, and modified it to use our optimized set of cover primitives (Figure 8a).

Note that our geometry shader outputs triangle strips rather than independent triangles (Figure 10). This results in a small speed improvement of 10–20%, and was applied in all measurements.

Line illustration Many rendering techniques enhance object appearance by emphasizing silhouettes or other important contours, e.g. [Gooch and Gooch 2001; DeCarlo et al. 2003]. With geometry shaders, silhouette rendering is in some sense an ideal scenario to highlight the efficiency of GPU edge-processing, because the output silhouette often has sub-linear complexity. We developed a simple prototype that renders both silhouettes and sharp features (Figure 8b). Again, we used as a baseline the basic approach of processing all faces as triangles with adjacency following the framework from the DX10 shadow volume sample. Since we use small quadrilaterals as the rendering primitive, we can control the thickness of the lines, as well as render the mesh triangles in white for hidden line removal, all in a single rendering pass.

Motion blur Wloka and Zeleznik [1996] describe a real-time technique to approximate the blur caused by the motion of an object relative to the viewer. The approach has similarities to shadow volumes, in that the mesh is also split into a front part and a rear part, and silhouette edges (with respect to the *motion*) are extruded to form quadrilaterals. As in the previous applications, we compared against the traditional approach of processing all triangles with adjacency. The resulting *motion volume* is rendered with partial transparency. Using a geometry shader program, this can all be done in a single rendering pass (Figure 8c).

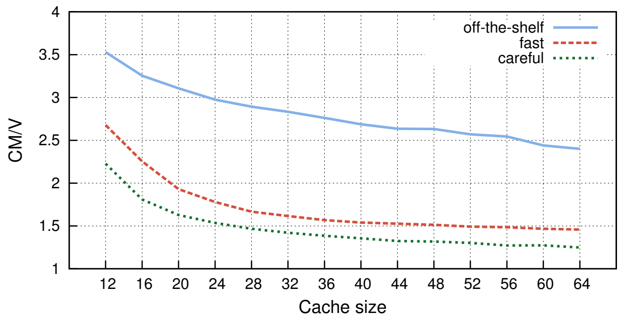
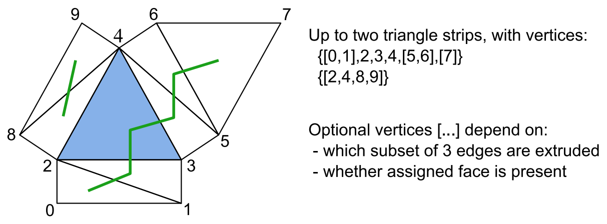


Figure 9: Number of cache misses per vertex comparing our ordering algorithms against the off-the-shelf technique, when run on the bunny mesh.



9
6
7
4
8
5
2
3
0
1
Up to two triangle strips, with vertices:
{[0,1],2,3,4,[5,6],[7]}
{[2,4,8,9]}
Optional vertices [...] depend on:
- which subset of 3 edges are extruded
- whether assigned face is present

Figure 10: Grouping of geometry shader output to form triangle strips, for the applications of shadow volume and motion blur. Just two strips are sufficient in all cases.

As in the shadow volume application, we output triangle strips since the extruded quadrilaterals share vertices with the emitted triangle faces (Figure 10).

Quantitative speedups We measured performance speedup factors of using our approach against the baseline methods outlined above for each application. Measurements were performed on both an NVIDIA GeForce 8800GTX and an AMD ATI Radeon HD2900. To simulate a more realistic graphics scene, the measurements used multiple instances of the models to factor out overhead due to frame setup and other elements in the scene.

Through experimentation, we observed best results for our applications on the AMD and NVIDIA cards when setting the vertex cache parameter to 12 and 24 vertices, respectively. At any rate, the results we report were not significantly affected by the cache size parameter. For our geometry-shader-bound prototype applications, which just use Gouraud shading, the vertex programs may be overly simple. We have verified that a more expensive vertex program does give rise to greater speedups. It is likely that geometry shader implementations may become more efficient in the future, thus further increasing the importance of careful primitive ordering.

The speedups produced by using our approach are reported in Table 2. Since we are processing just over half of the number of primitives, the speedup of the cover computation could reach close to 2×. However, in practice this does not occur because each of our geometry shader instances emit more data. In general, these applications results in speedups between 1.5× and 2.0× for most input meshes.

Other applications Many other applications would also benefit from efficient edge traversal. Some notable examples include fur rendering by selective extrusion of fins from edges near the silhouette [Lengyel et al. 2001], soft-shadows using penumbra wedges [Assarsson and Akenine-Möller 2003], and beveled edges [Bahnassi and Bahnassi 2007].

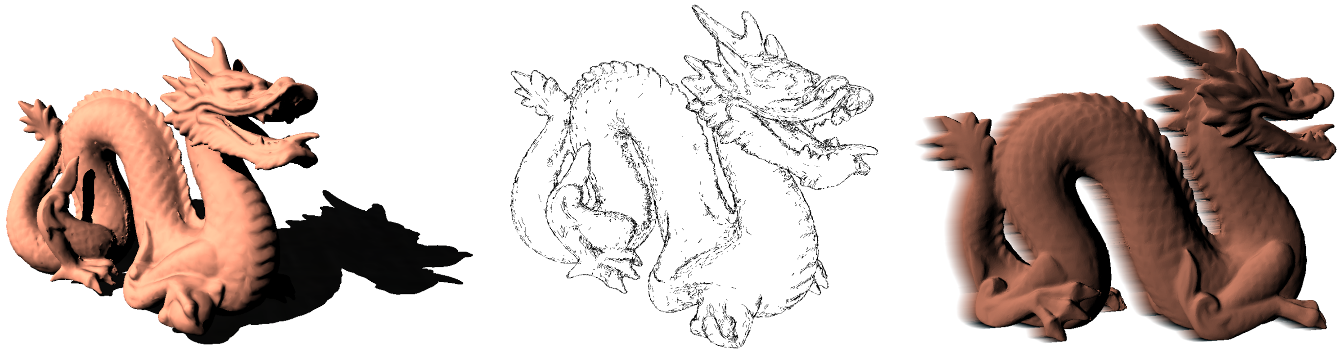

(a) Shadow volume (b) Line illustration (c) Motion blur

Figure 8: Screenshots of our prototype applications.

Table 2: Rendering speedup of our prototype applications compared to the common approach of processing one adjacency primitive per mesh face. Timings of our approach (in milliseconds) are shown in parenthesis.

Input mesh	Shadow volume		Line illustration		Motion blur	
Name	AMD	NVIDIA	AMD	NVIDIA	AMD	NVIDIA
fandisk	1.65	1.76 (0.6)	1.70	1.82 (0.6)	1.54	1.52 (0.6)
gargoyle	1.58	1.76 (1.2)	1.70	1.83 (1.2)	1.54	1.51 (1.1)
feline	1.60	1.77 (2.3)	1.72	1.87 (2.4)	1.59	1.52 (2.2)
bunny	1.73	1.76 (3.2)	1.81	1.92 (3.7)	1.60	1.48 (3.4)
dragon	1.74	1.82 (7.7)	1.84	1.89 (7.9)	1.62	1.52 (8.5)
turtle	1.70	1.75 (13.7)	1.80	1.96 (14.0)	1.60	1.48 (14.9)
buddha	1.70	1.73 (55.5)	1.80	1.95 (57.0)	1.61	1.50 (60.8)

One final note is that the set of cover faces can also be used to render the wireframe of the model (without a geometry shader, i.e. with mode `D3D10_FILL_WIREFRAME`). We noticed a speedup of roughly 70% when using the cover faces rather than all faces of the model. This is naturally not as efficient as the 100% improvement that can be obtained by creating a buffer of lines containing the edges of the model. However, it may be of practical value if the list of cover faces is already available (e.g., for one the above applications), since it would not consume any additional video memory.

8 Discussion

Dismissed alternative schemes We briefly summarize some other schemes that we considered and their associated drawbacks.

Each edge could be processed individually by considering its pair of adjacent faces, encoded by 4 vertices, e.g. in an indexed *line*-with-adjacency primitive. However, this would result in an index buffer with $4e$ indices for e edges, about twice as many as in our scheme. Also, face processing could not be well load-balanced among these primitives.

The triangle-*strip*-with-adjacency primitive would appear to be a promising way to string together several triangle-with-adjacency primitives. However, even on a regular mesh region, one cannot obtain a regular covering of the mesh edges. Therefore, branching would be necessary in the geometry shader, again degrading SIMD parallelism efficiency.

Limitations In current GPUs, the introduction of any geometry shader slows down the rendering pipeline. Therefore using our representation to render just the mesh faces incurs additional cost. However, for most applications that perform edge processing, the bottleneck is in the edge processing itself, and therefore the mesh rendering time does not significantly impact the results. The timing numbers in the previous section take that into account. Since geometry shaders are a novel architectural feature, they may be better optimized in future hardware generations.

9 Summary and future work

We have designed an efficient representation for processing both edges and faces of a mesh on a GPU. Our optimized traversal representation provides two separate improvements over prior approaches: (1) a nearly two-fold reduction in the number of adjacency primitives, and (2) additional reduction in vertex processing and memory bandwidth due to improved vertex caching. These optimizations result in substantial gains for several real-time rendering methods.

Future hardware is likely to include larger primitives in the form of surface patches. Defining efficient traversals for such patches will be an interesting area of continuing research.

Acknowledgments

Pedro Sander was partly supported by Hong Kong RGC CERG grant #619008.

References

ANDRADE, D., RESENDE, M. G. C., and WERNECK, R. 2008. Fast local search for the maximum independent set problem. In *Proceedings of Workshop on Experimental Algorithms, LNCS 5038*, pages 220–234.

ASSARSSON, U. and AKENINE-MÖLLER, T. 2003. A geometry-based soft shadow volume algorithm using graphics hardware. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH 2003)*, 22(3):511–520.

BAHNASSI, H. and BAHNASSI, W. 2007. Micro-beveled edges. In *ShaderX5: Advanced Rendering Techniques*. Charles River Media.

BLYTHE, D. 2006. The Direct3D 10 system. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH 2003)*, 25(3):724–734.

BRENNAN, C. 2002. Shadow volume extrusion using a vertex shader. In *ShaderX: Vertex and Pixel Shader Tips and Tricks*. Wordware.

CARD, D. and MITCHELL, J. 2002. Non-photorealistic rendering with pixel and vertex shaders. In *ShaderX: Vertex and Pixel Shader Tips and Tricks*. Wordware.

CHHUGANI, J. and KUMAR, S. 2007. Geometry engine optimization: cache friendly compressed representation of geometry. In *Proceedings of Symposium on Interactive 3D Graphics and Games (I3D)*, pages 9–16.

CHOW, M. M. 1997. Optimized geometry compression for real-time rendering. In *IEEE Visualization*, pages 347–354.

CROW, F. C. 1977. Shadow algorithms for computer graphics. In *Proceedings of ACM SIGGRAPH 77*, pages 242–248.

DECARLO, D., FINKELSTEIN, A., RUSINKIEWICZ, S., and SANTELLA, A. 2003. Suggestive contours for conveying shape. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH 2003)*, 22(3):848–855.

DEERING, M. 1995. Geometry compression. In *Proceedings of ACM SIGGRAPH 95*, pages 13–20.

EDMONDS, J. and JOHNSON, E. L. 1973. Matching, Euler tours and the Chinese postman. *Mathematical Programming*, 5:88–129.

ESTKOWSKI, R., MITCHELL, J. S. B., and XIANG, X. 2002. Optimal decomposition of polygonal models into triangle strips. In *Proceedings of Symposium on Computational Geometry*, pages 254–263.

EVANS, F., SKIENA, S., and VARSHNEY, A. 1996. Optimizing triangle strips for fast rendering. In *IEEE Visualization*, pages 319–326.

GABOW, H. N. 1974. *Implementation of Algorithms for Maximum Matching on Nonbipartite Graphs*. PhD thesis, Stanford University.

GAREY, M. R. and JOHNSON, D. S. 1977. The rectilinear Steiner tree problem is *NP*-complete. *SIAM Journal on Applied Mathematics*, 32(4):826–834.

GOOCH, B. and GOOCH, A. 2001. *Non-photorealistic rendering*. A. K. Peters, Ltd.

GROSSO, A., LOCATELLI, M., and PULLAN, W. 2007. Simple ingredients leading to very efficient heuristics for the maximum clique problem. *Journal of Heuristics*, on-line.

HALL, P. 1935. On representatives of subsets. *Journal of the London Mathematical Society*, 10:26–30.

HEIDMANN, T. 1991. Real shadows real time. *Iris Universe*, 18: 28–31.

HERTZMANN, A. 1999. Silhouettes and outlines. In *Introduction to 3D Non-Photorealistic Rendering*, chapter 7. ACM SIGGRAPH Course Notes.

HOPCROFT, J. E. and KARP, R. M. 1973. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM Journal on Computing*, 2(4):225–231.

HOPPE, H. 1999. Optimization of mesh locality for transparent vertex caching. In *Proceedings of ACM SIGGRAPH 99*, pages 269–276.

KARYPIS, G. and KUMAR, V. 1995. *METIS: Unstructured Graph Partitioning and Sparse Matrix Ordering System, Version 2.0.*

LENGYEL, J., PRAUN, E., FINKELSTEIN, A., and HOPPE, H. 2001. Real-time fur over arbitrary surfaces. In *Proceedings of Symposium on Interactive 3D Graphics and Games (I3D)*, pages 227–232.

LIN, G. and YU, T. P.-Y. 2006. An improved vertex caching scheme for 3D mesh rendering. *IEEE Transactions on Visualization and Computer Graphics*, 12(4):640–648.

MCGUIRE, M. and HUGHES, J. 2004. Hardware-determined feature edges. In *Proceedings of Symposium on Non-Photorealistic Animation and Rendering (NPAR)*, pages 35–47.

MICROSOFT CORP. 2007. *DirectX 10 SDK*.

ROTHBERG, E. 1985. MATHPROG. `http://elib.zib.de/pub/Packages/mathprog/matching/weighted`.

SANDER, P. V., NEHAB, D., and BARCZAK, J. 2007. Fast triangle reordering for vertex locality and reduced overdraw. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH 2007)*, 26(3):89.

TARIQ, S. 2007. Fur (using shells and fins). Technical Report WP-03021-001-v01, NVIDIA Corp.

WLOKA, M. M. and ZELEZNIK, R. C. 1996. Interactive real-time motion blur. *The Visual Computer*, 12(6):283–295.

XIANG, X., HELD, M., and MITCHELL, J. S. B. 1999. Fast and effective stripification of polygonal surface models. In *Proceedings of Symposium on Interactive 3D Graphics and Games (I3D)*, pages 71–78.

Appendix: Finding a lower bound on $|C|$

A simple lower bound on the size $|C|$ of our vertex cover is obtained as follows. Let $d(v)$ denote the degree of vertex v (in either the primal or the dual graph). Then, $d(v') \leq 3$ for every dual vertex $v' \in V'$ since each triangle has at most 3 neighbors. Thus, every vertex in C covers at most three edges, and so $|C| \geq |E'|/3$.

This can be tightened by obtaining a lower bound on the number of edges that must be covered *twice*. The intuition is that all odd-degree vertices in the mesh must have at least one adjacent edge that is covered twice, and moreover these doubly covered edges must form paths that terminate only at other odd-degree vertices.

Formally, consider a vertex v in the primal graph, the ring $R(v)$ of dual vertices surrounding v (i.e. a red cycle in Figure 3a), and let $t(v)$ denote the number of edges in $R(v)$ that are covered twice. Since C is a cover, one can show that $t(v) \equiv d(v) \pmod 2$. Let $\tilde{G} = (V, \tilde{E})$ be a minimum subgraph of G (minimizing $|\tilde{E}|$) such that for all $v \in V$, $d_{\tilde{G}}(v) \equiv d_G(v) \pmod 2$. Then the number of edges covered twice by C must be at least $|\tilde{E}|$. Arguing as before, we obtain $|C| \geq (|E'| + |\tilde{E}|)/3$.

A set of edges for which the odd-degree vertices coincide with a set T is called a T-join. Thus, the set $\tilde{E}$ is a minimum V_{odd}-join, where V_{odd} is the set of odd-degree vertices in G.

It has been shown that finding a minimum T-join can be reduced to a minimum-weight perfect matching problem. In particular, Edmonds and Johnson [1973] have shown that the minimum V_{odd}-join is equivalent to a minimum weight matching between odd-degree vertices, where the weight attached to any pair of vertices $u, v \in V_{\text{odd}}$ is exactly the shortest-path distance in G, $\text{dist}_G(u, v)$.

Thus, to compute $|\tilde{E}|$, we find this minimum-weight perfect matching using the $O(|V_{\text{odd}}|^3)$ algorithm by Gabow [1974], as implemented in the METIS library [Karypis and Kumar 1995]. We are able to greatly speed up this computation by noting that a shortest path between two vertices in V_{odd} that crosses two other vertices in V_{odd} cannot contribute to the minimum-weight perfect matching. We call such paths *illegal*. Often this prunes 99% of the search space. To verify the validity of this optimization, we prove the following lemma (the actual proof is not necessary for the implementation of either our algorithm, or the lower bound).

Lemma 2. *Let $G = (V, E)$ and $\tilde{G} = (V, \tilde{E})$ where $\tilde{E}$ is a minimum V_{odd}-join. Then*

1. *$\tilde{E}$ is a disjoint union of legal paths between vertices in V_{odd}.*
2. *Let $G_{\mathrm{prune}} = (V_{\mathrm{odd}}, E_{\mathrm{prune}})$ be a weighted graph where we connect $u, v \in V_{\mathrm{odd}}$ by an edge of weight $\mathrm{dist}_G(u, v)$ if and only if no shortest path between u and v is illegal. Then any G_{prune} has a perfect matching and any minimum-weight perfect matching has weight $|\tilde{E}|$.*

Proof. To prove (1), note that $\tilde{G}$ contains no cycles (their removal preserves a V_{odd}-join). Thus, $\tilde{G}$ is a *forest*, i.e. a vertex-disjoint union of trees. For each such tree T, we find a legal path decomposition by repeatedly performing the following: Contract all uninterrupted paths in T to individual edges. Choose a lowest leaf and match it to its sibling, removing the connecting path in T.

To prove (2), note that for any such matching, if the total length of all shortest paths is minimal, the paths must be edge-disjoint. Thus, the union of all paths is indeed a V_{odd}-join. We must show that for some minimal V_{odd}-join $\tilde{E}$, no $u, v \in V_{\mathrm{odd}}$ connected by a legal path in $\tilde{E}$ are also connected by an illegal shortest path. We start with two crucial observations, which follow from the minimality of $\tilde{E}$:

Observation 1. *For any vertices a, b in the same tree T in $\tilde{G}$, the shortest tree path $T_{a,b}$ connecting them is a shortest path in G.*

Observation 2. *For any vertices a, b as above, the tree path $T_{a,b}$ is the only shortest path that intersects $\tilde{E}$.*

Both observations follow from the same principle. If we had a path p^* between a and b that violates either Observation 1 or Observation 2 (that is, p^* is shorter than $T_{a,b}$ or is a distinct shortest path intersecting $\tilde{E}$), we could construct a strictly smaller V_{odd}-join by removing path $T_{a,b}$ and taking the symmetric difference of the remaining edge set and p^*. That is, the set $(\tilde{E} \setminus T_{a,b}) \setminus p^* \cup p^* \setminus (\tilde{E} \setminus T_{a,b})$ would contradict the minimality of $\tilde{E}$ (see replacements in Figure 11).

We may assume that $\tilde{G}$ contains the minimum number of trees among all minimum V_{odd}-joins. Suppose for the sake of contradiction that for some vertices $u, v \in V_{\mathrm{odd}}$ in some tree T connected by a legal T-path p, there exists an illegal shortest path p' containing two internal vertices $x, y \in V_{\mathrm{odd}}$. We consider two cases:

Case 1: Nodes x and y are in the same tree T' (possibly $T' = T$) in $\tilde{G}$ (Figure 11a). *Proof.* Let $p'_{x,y}$ be the sub-path of p' from x to y, and let $T'_{x,y}$ be the shortest tree path from x to y. By the minimality of p' and Observation 1, both paths are shortest paths from x to y. Thus by replacing $p'_{x,y}$ with $T'_{x,y}$, we get a shortest path $p'' = p' \setminus p'_{x,y} \cup T'_{x,y}$ (Figure 11b). This path is illegal, and so distinct from p, and it intersects $\tilde{E}$, which by Observation 2 contradicts the minimality of $\tilde{E}$ (Figure 11c).

Case 2: For trees $T' \neq T''$ in $\tilde{G}$, $x \in T'$ and $y \in T''$ (Figure 11d). *Proof.* By Observation 2, p' does not intersect $\tilde{E}$. Thus if we replace p with p', the new V_{odd}-join $\tilde{E}' = (\tilde{E} \setminus p) \cup p'$ (Figure 11e) contains strictly fewer trees than $\tilde{E}$, contradicting our assumption for $\tilde{G}$. □

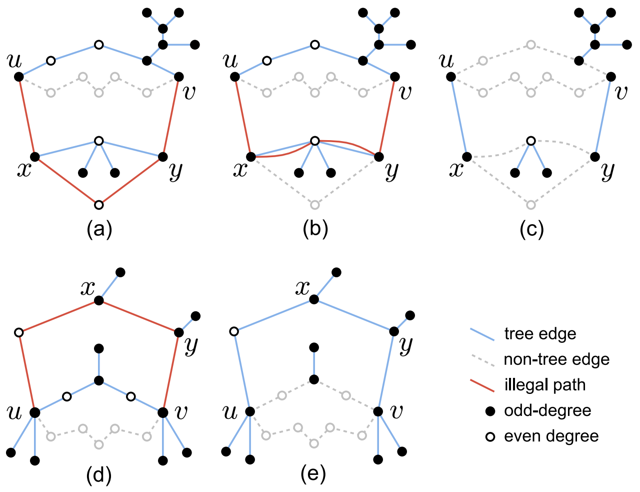


Figure 11: (a) Case 1. (b) Collapse p' to T'. (c) Replacement yields fewer edges. (d) Case 2. (e) Replacement yields fewer trees.