A tree traversal algorithm for decision problems in knot theory and 3-manifold topology

Benjamin A. Burton and Melih Ozlen

Abstract

In low-dimensional topology, many important decision algorithms are based on normal surface enumeration, which is a form of vertex enumeration over a high-dimensional and highly degenerate polytope. Because this enumeration is subject to extra combinatorial constraints, the only practical algorithms to date have been variants of the classical double description method. In this paper we present the first practical normal surface enumeration algorithm that breaks out of the double description paradigm. This new algorithm is based on a tree traversal with feasibility and domination tests, and it enjoys a number of advantages over the double description method: incremental output, significantly lower time and space complexity, and a natural suitability for parallelisation. Experimental comparisons of running times are included.

AMS Classification Primary 57N10, 52B55; Secondary 90C05, 57N35

Keywords Normal surfaces, vertex enumeration, tree traversal, backtracking, linear programming

1 Introduction

Much research in low-dimensional topology has been driven by decision problems. Examples from knot theory include *unknot recognition* (given a polygonal representation of a knot, decide whether it is equivalent to a trivial unknotted loop), and the more general problem of *knot equivalence* (given two polygonal representations of knots, decide whether they are equivalent to each other). Analogous examples from 3-manifold topology include *3-sphere recognition* (given a triangulation of a 3-manifold, decide whether this 3-manifold is the 3-dimensional sphere), and the more general *homeomorphism problem* (given two triangulations of 3-manifolds, decide whether these 3-manifolds are homeomorphic, i.e., "topologically equivalent").

Significant progress has been made on these problems over the past 50 years. For instance, Haken introduced an algorithm for recognising the unknot in the 1960s [22], and Rubinstein presented the first 3-sphere recognition algorithm in the early 1990s [35]. Since Perelman's recent proof of the geometrisation conjecture [29], we now have a complete (but extremely complex) algorithm for the homeomorphism problem, pieced together though a diverse array of techniques by many different authors [25, 31].

A key problem remains with many 3-dimensional decision algorithms (including all of those mentioned above): the best known algorithms run in exponential time (and for some problems, worse), where the input size is measured by the number of crossings in a knot diagram, or the number of tetrahedra in a 3-manifold triangulation. This severely limits the

practicality of such algorithms. For instance, it took around 30 years to resolve Thurston's well-known conjecture regarding the Weber-Seifert space [6]—although an algorithmic solution was known in the 1980s [27], it took another 25 years of development before the necessary computations could be performed [14]. Here the input size was $n = 23$.

The most successful machinery for 3-dimensional decision problems has been *normal surface theory*. Normal surfaces were introduced by Kneser [30] and later developed by Haken for algorithmic use [22, 23], and they play a key role in all of the decision algorithms mentioned above. In essence, they allow us to convert difficult "topological searches" into a "combinatorial search" in which we enumerate vertices of a high-dimensional polytope called the *projective solution space*.[1] In Section 2 we outline the combinatorial aspects of the projective solution space; see [24] for a broader introduction to normal surface theory in a topological setting.

For many topological decision problems, the computational bottleneck is precisely the vertex enumeration described above. Vertex enumeration over a polytope is a well-studied problem, and several families of algorithms appear in the literature. These include *backtracking algorithms* [4, 19], pivot-based *reverse search algorithms* [1, 3, 18], and the inductive *double description method* [34]. All have worst-case running times that are super-polynomial in both the input *and* the output size, and it is not yet known whether a polynomial time algorithm (with respect to the combined input and output size) exists.[2] For topological problems, the corresponding running times are exponential in the "topological input size"; that is, the number of crossings or tetrahedra.

Normal surface theory poses particular challenges for vertex enumeration:

- The projective solution space is a highly degenerate polytope (that is, a d-dimensional polytope in which vertices typically belong to significantly more than d facets).
- Exact arithmetic is required. This is because each vertex of the projective solution space is a rational vector that effectively "encodes" a topological surface, and topological decision algorithms require us to scale each vertex to its smallest integer multiple in order to extract the relevant topological information.
- We are not interested in *all* of the vertices of the projective solution space, but just those that satisfy a powerful set of conditions called the *quadrilateral constraints*. These are extra combinatorial constraints that eliminate all but a small fraction of the polytope, and an effective enumeration algorithm should be able to enforce them as it goes, not simply use them as a post-processing output filter.

All well-known implementations of normal surface enumeration [8, 16] are based on the double description method: degeneracy and exact arithmetic do not pose any difficulties, and it is simple to embed the quadrilateral constraints directly into the algorithm [12]. Section 2 includes a discussion of why backtracking and reverse search algorithms have not been used to date. Nevertheless, the double description method does suffer from significant drawbacks:

- The double description method can suffer from a *combinatorial explosion*: even if both the input and output sizes are small, it can create extremely complex intermediate polytopes where the number of vertices is super-polynomial in both the input and output sizes. This in turn leads to an unwelcome explosion in both time and memory

[1] For some topological algorithms, vertex enumeration is not enough: instead we must enumerate a Hilbert basis for a polyhedral cone.

[2] Efficient algorithms do exist for certain classes of polytopes. For example, in the case of non-degenerate polytopes, the reverse search algorithm of Avis and Fukuda has virtually no space requirements beyond storing the input, and has a running time polynomial in the combined input and output size [3].

use, a particularly frustrating situation for normal surface enumeration where the output size is often small (though still exponential in general) [10].

- It is difficult to parallelise the double description method, due to its inductive nature. There have been recent efforts in this direction [37], though they rely on a shared memory model, and for some polytopes the speed-up factor plateaus when the number of processors grows large.
- Because of its inductive nature, the double description method typically does not output *any* vertices until the entire algorithm is complete. This effectively removes the possibility of early termination (which is desirable for many topological algorithms), and it further impedes any attempts at parallelisation.

In this paper we present a new algorithm for normal surface enumeration. This algorithm belongs to the backtracking family, and is described in detail in Section 3. Globally, the algorithm is structured as a tree traversal, where the underlying tree is a decision tree indicating which coordinates are zero and which are non-zero. Locally, we move through the tree by performing incremental feasibility tests using the dual simplex method (though interior point methods could equally well be used).

Fukuda et al. [19] discuss an impediment to backtracking algorithms, which is the need to solve the NP-complete *restricted vertex problem*. We circumvent this by ordering the tree traversal in a careful way and introducing *domination tests* over the set of previously-found vertices. This introduces super-polynomial time and space trade-offs, but for normal surface enumeration these trade-offs are typically not severe (we quantify this both theoretically and empirically within Sections 5 and 6 respectively).

Our algorithm is well-suited to normal surface enumeration. It integrates the quadrilateral constraints seamlessly into the tree structure. Moreover, we are often able to perform exact arithmetic using native machine integer types (instead of arbitrary precision integers, which are significantly slower). We do this by exploiting the sparseness of the equations that define the projective solution space, and thereby bounding the integers that appear in intermediate calculations. The details appear in Section 4.

Most significantly, this new algorithm is the first normal surface enumeration algorithm to break out of the double description paradigm. Furthermore, it enjoys a number of advantages over previous algorithms:

- The theoretical worst-case bounds on both time and space complexity are significantly better for the new algorithm. In particular, the space complexity is a small polynomial in the combined input and output size. See Section 5 for details.
- The new algorithm lends itself well to parallelisation, including both shared memory models and distributed processing, with minimal need for inter-communication and synchronisation.
- The new algorithm produces incremental output, which makes it well-suited for early termination and further parallelisation.
- The tree traversal supports a natural sense of "progress tracking", so that users can gain a rough sense for how far they are through the enumeration procedure.

In Section 6 we measure the performance of the new algorithm against the prior state of the art, using a rich test suite with hundreds of problems that span a wide range of difficulties. For simple problems we find the new algorithm slower, but as the problems become more difficult the new algorithm quickly outperforms the old, often running orders of magnitude faster. We conclude that for difficult problems this new tree traversal algorithm is superior, owing to both its stronger practical performance and its desirable theoretical attributes as outlined above.

2 Preliminaries

For decision problems based on normal surface theory, the input is typically a *3-manifold triangulation*. This is a collection of n tetrahedra where some or all of the $4n$ faces are affinely identified (or "glued together") in pairs, so that the resulting topological space is a 3-manifold (possibly with boundary). The *size of the input* is measured by the number of tetrahedra, and we refer to this quantity as n throughout this paper.

We allow two faces of the same tetrahedron to be identified together; likewise, we allow different edges or vertices of the same tetrahedron to be identified. Some authors refer to such triangulations as *generalised triangulations*. In essence, we allow the tetrahedra to be "bent" or "twisted", instead of insisting that they be rigidly embedded in some larger space. This more flexible definition allows us to keep the input size n small, which is important when dealing with exponential algorithms.

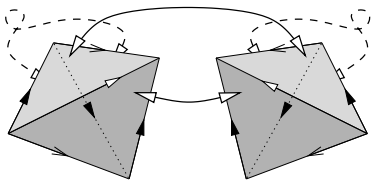

Figure 1: A 3-manifold triangulation of size $n = 2$

Figure 1 illustrates a triangulation with $n = 2$ tetrahedra: the back two faces of each tetrahedron are identified with a twist, and the front two faces of the first tetrahedron are identified with the front two faces of the second tetrahedron. As a consequence, all eight tetrahedron vertices become identified together, and the edges become identified into three equivalence classes as indicated by the three different arrowheads. We say that the overall triangulation has *one vertex* and *three edges*. The underlying 3-manifold represented by this triangulation is the product space $S^2 \times S^1$.

A *closed* triangulation is one in which all $4n$ tetrahedron faces are identified in $2n$ pairs (as in the example above). A *bounded* triangulation is one in which some of the $4n$ tetrahedron faces are left unidentified; together these unidentified faces form the *boundary* of the triangulation. The underlying topological spaces for closed and bounded triangulations are closed and bounded 3-manifolds respectively.

There are of course other ways of representing a 3-manifold (for instance, Heegaard splittings or Dehn surgeries). We use triangulations because they are "universal": it is typically easy to convert some other representation into a triangulation, whereas it can sometimes be difficult to move in the other direction. In particular, when we are solving problems in knot theory, we typically work with a triangulation of the *knot complement*—that is, the 3-dimensional space surrounding the knot.

In normal surface theory, interesting surfaces within a 3-manifold triangulation can be encoded as integer points in $\mathbb{R}^{3n}$. Each such point $\mathbf{x} \in \mathbb{R}^{3n}$ satisfies the following conditions:

- $\mathbf{x} \geq \mathbf{0}$ and $M\mathbf{x} = \mathbf{0}$, where M is a matrix of *matching equations* that depends on the input triangulation;
- $\mathbf{x}$ satisfies the *quadrilateral constraints*: for each triple of coordinates (x_1, x_2, x_3), (x_4, x_5, x_6), …, $(x_{3n-2}, x_{3n-1}, x_{3n})$, at most one of the three entries in the triple can be non-zero. There are n such triples, giving n such restrictions in total.

Any point $\mathbf{x} \in \mathbb{R}^{3n}$ that satisfies all of these conditions (that is, $\mathbf{x} \geq \mathbf{0}$, $M\mathbf{x} = \mathbf{0}$ and the quadrilateral constraints) is called *admissible*.

It should be noted that normal surface theory is often described using a different coordinate system ($\mathbb{R}^{7n}$, also known as *standard coordinates*). Our coordinates in $\mathbb{R}^{3n}$ are known as *quadrilateral coordinates*, and were introduced by Tollefson [39]. They are ideal for computation because they are smaller and faster, and because information that is "lost" from standard coordinates is typically quick to reconstruct [9].

These coordinates have a natural geometric interpretation, which we outline briefly. Typically one can arrange for an interesting surface within the 3-manifold to intersect each tetrahedron of the triangulation in a collection of disjoint triangles and / or quadrilaterals, as illustrated in Figure 2. The corresponding admissible integer point in $\mathbb{R}^{3n}$ counts how many quadrilaterals slice through each tetrahedron in each of three possible directions. From this information, the locations of the triangles can also be reconstructed, under some weak assumptions about the surface.

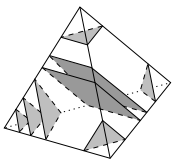

Figure 2: Triangles and quadrilaterals in a tetrahedron

The matching equations ensure that triangles and quadrilaterals in adjacent tetrahedra can be joined together, and the quadrilateral constraints ensure that the resulting surface does not intersect itself. We do not use this geometric interpretation in this paper, and we do not discuss it further; for details the reader is referred to [24].

We recount some well-known facts about the matching equations [9, 38, 39]:

Lemma 2.1. *Let e and v be the number of edges and vertices of the triangulation that do not lie within the boundary. Then there are e matching equations; that is, M is an $e \times 3n$ matrix. In the case of a closed triangulation, we have $e = n + v > n$.*

In general one or more matching equations may be redundant. In the case of a closed triangulation, the rank of M is precisely n.

Lemma 2.2. *The matrix M is sparse with small integer entries. Each column contains at most four non-zero entries, each of which is ± 1, ± 2, ± 3 or ± 4. Moreover, the sum of absolute values in each column is at most 4; that is, $\sum_i |M_{i,c}| \leq 4$ for all c. If the triangulation is orientable, the only possible non-zero entries are ± 1 and ± 2.*

We define the *solution cone* $\mathscr{Q}^\vee$ to be the set

$$\mathscr{Q}^\vee = \left\{ \mathbf{x} \in \mathbb{R}^{3n} \;\middle|\; \mathbf{x} \geq \mathbf{0} \text{ and } M\mathbf{x} = \mathbf{0} \right\}.$$

This is a polyhedral cone with apex at the origin. From above, it follows that interesting surfaces within our 3-manifold are encoded as integer points within the solution cone. The *projective solution space* $\mathscr{Q}$ is defined to be a cross-section of this cone:

$$\mathscr{Q} = \left\{ \mathbf{x} \in \mathscr{Q}^\vee \;\middle|\; \sum x_i = 1 \right\}.$$

It follows that the projective solution space is a (bounded) polytope with rational vertices. In general, both the solution cone and the projective solution space are highly degenerate (that is, extreme rays of $\mathscr{Q}^\vee$ and vertices of $\mathscr{Q}$ often belong to many more facets than required by the dimension).

For polytopes and polyhedra, we follow the terminology of Ziegler [40]: polytopes are always bounded, polyhedra may be either bounded or unbounded, and all polytopes and polyhedra are assumed to be convex.

It is important to note that the quadrilateral constraints do not feature in the definitions of $\mathscr{Q}^\vee$ or $\mathscr{Q}$. These constraints are combinatorial in nature, and they cause significant theoretical problems: they are neither linear nor convex, and the region of $\mathscr{Q}$ that obeys them can be disconnected or can have non-trivial topology (such as "holes"). Nevertheless, they are extremely powerful: although the vertices of $\mathscr{Q}$ can be numerous, typically just a tiny fraction of these vertices satisfy the quadrilateral constraints [10, 13].

For many topological decision problems that use normal surface theory, a typical algorithm runs as follows:

1. Enumerate all admissible vertices of the projective solution space $\mathscr{Q}$ (that is, all vertices of $\mathscr{Q}$ that satisfy the quadrilateral constraints);
2. Scale each vertex to its smallest integer multiple, "decode" it to obtain a surface within the underlying 3-manifold triangulation, and test whether this surface is "interesting";
3. Terminate the algorithm once an interesting surface is found, or once all vertices are exhausted.

The definition of "interesting" varies for different decision problems. For more of the overall topological context see [24], and for more details on quadrilateral coordinates and the projective solution space see [9].

As indicated in the introduction, step 1 (the enumeration of vertices) is typically the computational bottleneck. Because so few vertices are admissible, it is crucial for normal surface enumeration algorithms to enforce the quadrilateral constraints as they go—one cannot afford to enumerate the entire vertex set of $\mathscr{Q}$ (which is generally orders of magnitude larger) and then enforce the quadrilateral constraints afterwards.

Traditionally, normal surface enumeration algorithms are based on the double description method of Motzkin et al. [34], with additional algorithmic improvements specific to normal surface theory [12]. In brief, we construct a sequence of polytopes $P_0, P_1, \ldots, P_e$, where P_0 is the unit simplex in $\mathbb{R}^{3n}$, P_e is the final solution space $\mathscr{Q}$, and each intermediate polytope P_i is the intersection of the unit simplex with the first i matching equations. The algorithm works inductively, constructing P_{i+1} from P_i at each stage. For each intermediate polytope P_i we throw away any vertices that do not satisfy the quadrilateral constraints, and it can be shown inductively that this yields the correct final solution set.

It is worth noting why backtracking and reverse search algorithms have not been used for normal surface enumeration to date:

- Reverse search algorithms map out vertices by pivoting along edges of the polytope. This makes it difficult to incorporate the quadrilateral constraints: since the region that satisfies these constraints can be disconnected, we may need to pivot through vertices that *break* these constraints in order to map out all solutions. Degeneracy can also cause significant performance problems for reverse search algorithms [1, 2].
- Backtracking algorithms receive comparatively little attention in the literature. Fukuda et al. [19] show that backtracking can solve the *face* enumeration problem in polynomial time. However, for vertex enumeration their framework requires a solution to the NP-complete *restricted vertex problem*, and they conclude that straightforward backtracking is unlikely to work efficiently for vertex enumeration in general.

Our new algorithm belongs to the backtracking family. Despite the concerns of Fukuda et al., we find in Section 6 that for large problems our algorithm is a significant improvement

over the prior state of the art, often running orders of magnitude faster. We achieve this by introducing *domination tests*, which are time and space trade-offs that allow us to avoid solving the restricted vertex problem directly. The full algorithm is given in Section 3 below, and in Sections 5 and 6 we study these trade-offs both theoretically and experimentally.

3 The tree traversal algorithm

The basic idea behind the algorithm is to construct admissible vertices $\mathbf{x} \in \mathscr{Q}$ by iterating through all possible combinations of which coordinates x_i are zero and which coordinates x_i are non-zero. We arrange these combinations into a search tree of height n, which we traverse in a depth-first manner. Using a combination of feasibility tests and domination tests, we are able to prune this search tree so that the traversal is more efficient, and so that the leaves at depth n correspond precisely to the admissible vertices of $\mathscr{Q}$.

Because the quadrilateral constraints are expressed purely in terms of zero versus non-zero coordinates, we can easily build the quadrilateral constraints directly into the structure of the search tree. We do this with the help of *type vectors*, which we introduce in Section 3.1. In Section 3.2 we describe the search tree and present the overall structure of the algorithm, and we follow in Sections 3.3 and 3.4 with details on some of the more complex steps.

3.1 Type vectors

Recall the quadrilateral constraints, which state that for each $i = 1, \ldots, n$, at most one of the three coordinates $x_{3i-2}, x_{3i-1}, x_{3i}$ can be non-zero. A *type vector* is a sequence of n symbols that indicates how these constraints are resolved for each i. In detail:

Definition 3.1 (Type vector)**.** Let $\mathbf{x} = (x_1, \ldots, x_{3n}) \in \mathbb{R}^{3n}$ be any vector that satisfies the quadrilateral constraints. We define the *type vector* of $\mathbf{x}$ to be $\tau(\mathbf{x}) = (\tau_1, \ldots, \tau_n) \in \{0, 1, 2, 3\}^n$, where

$$\tau_i = \begin{cases} 0 & \text{if } x_{3i-2} = x_{3i-1} = x_{3i} = 0; \\ 1 & \text{if } x_{3i-2} \neq 0 \text{ and } x_{3i-1} = x_{3i} = 0; \\ 2 & \text{if } x_{3i-1} \neq 0 \text{ and } x_{3i-2} = x_{3i} = 0; \\ 3 & \text{if } x_{3i} \neq 0 \text{ and } x_{3i-2} = x_{3i-1} = 0. \end{cases} \tag{3.1}$$

For example, consider the vector $\mathbf{x} = (0, 1, 0,\ 0, 0, 4,\ 0, 0, 0,\ 0, 0, 2) \in \mathbb{R}^{12}$ where $n = 4$. Here the type vector of $\mathbf{x}$ is $\tau(\mathbf{x}) = (2, 3, 0, 3)$.

An important feature of type vectors is that they carry enough information to completely reconstruct any vertex of the projective solution space, as shown by the following result.

Lemma 3.2. *Let $\mathbf{x}$ be any admissible vertex of $\mathscr{Q}$. Then the precise coordinates $x_1, \ldots, x_{3n}$ can be recovered from the type vector $\tau(\mathbf{x})$ by solving the following simultaneous equations:*

- *the matching equations $M\mathbf{x} = \mathbf{0}$;*
- *the projective equation $\sum x_i = 1$;*
- *the equations of the form $x_j = 0$ as dictated by (3.1) above, according to the particular value (0, 1, 2 or 3) of each entry in the type vector $\tau(\mathbf{x})$.*

Proof. This result simply translates Lemma 4.4 of [12] into the language of type vectors. However, it is also a simple consequence of polytope theory: the type vector indicates which facets of $\mathscr{Q}$ the point $\mathbf{x}$ belongs to, and any vertex of a polytope can be reconstructed as the intersection of those facets to which it belongs. See [12] for further details. □

Note that this full recovery of $\mathbf{x}$ from $\tau(\mathbf{x})$ is only possible for *vertices* of $\mathscr{Q}$, not arbitrary points of $\mathscr{Q}$. In general, the type vector $\tau(\mathbf{x})$ carries only enough information for us to determine the smallest face of $\mathscr{Q}$ to which $\mathbf{x}$ belongs.

However, type vectors do carry enough information for us to identify *which* admissible points $\mathbf{x} \in \mathscr{Q}$ are vertices of $\mathscr{Q}$. For this we introduce the concept of domination.

Definition 3.3 (Domination)**.** Let $\tau, \sigma \in \{0,1,2,3\}^n$ be type vectors. We say that τ *dominates* σ if, for each $i = 1, \ldots, n$, either $\sigma_i = \tau_i$ or $\sigma_i = 0$. We write this as $\tau \geq \sigma$.

For example, if $n = 4$ then $(1,0,2,3) \geq (1,0,2,0) \geq (1,0,2,0) \geq (1,0,0,0)$. On the other hand, neither of $(1,0,2,0)$ or $(1,0,3,0)$ dominates the other. It is clear that in general, domination imposes a partial order (but not a total order) on type vectors.

Remark. We use the word "domination" because domination of type vectors corresponds precisely to domination in the face lattice of the polytope $\mathscr{Q}$. For any two admissible points $\mathbf{x}, \mathbf{y} \in \mathbb{R}^{3n}$, we see from Definition 3.1 that $\tau(\mathbf{x}) \geq \tau(\mathbf{y})$ if and only if, for every coordinate position i where $x_i = 0$, we also have $y_i = 0$. Phrased in the language of polytopes, this means that $\tau(\mathbf{x}) \geq \tau(\mathbf{y})$ if and only if every face of $\mathscr{Q}$ containing $\mathbf{x}$ also contains $\mathbf{y}$. That is, $\tau(\mathbf{x}) \geq \tau(\mathbf{y})$ if and only if the smallest-dimensional face containing $\mathbf{y}$ is a *subface* of the smallest-dimensional face containing $\mathbf{x}$.

These properties allow us to characterise admissible vertices of the projective solution space entirely in terms of type vectors, as seen in the following result.

Lemma 3.4. *Let $\mathbf{x} \in \mathscr{Q}$ be admissible. Then the following statements are equivalent:*

- $\mathbf{x}$ *is a vertex of $\mathscr{Q}$;*
- *there is no admissible point $\mathbf{y} \in \mathscr{Q}$ for which $\tau(\mathbf{x}) \geq \tau(\mathbf{y})$ and $\mathbf{x} \neq \mathbf{y}$;*
- *there is no admissible vertex $\mathbf{y}$ of $\mathscr{Q}$ for which $\tau(\mathbf{x}) \geq \tau(\mathbf{y})$ and $\mathbf{x} \neq \mathbf{y}$.*

Proof. This is essentially Lemma 4.3 of [9], whose proof involves elementary polytope theory with some minor complications due to the quadrilateral constraints. Here we merely reformulate this result in terms of type vectors. See [9] for further details. □

Because our algorithm involves backtracking, it spends much of its time working with partially-constructed type vectors. It is therefore useful to formalise this notion.

Definition 3.5 (Partial type vector)**.** A *partial type vector* is any vector $\tau = (\tau_1, \ldots, \tau_n)$, where each $\tau_i \in \{0,1,2,3,-\}$. We call the special symbol "$-$" the *unknown symbol*. A partial type vector that does not contain any unknown symbols is also referred to as a *complete type vector*.

We say that two partial type vectors τ and σ *match* if, for each $i = 1, \ldots, n$, either $\tau_i = \sigma_i$ or at least one of τ_i, σ_i is the unknown symbol.

For example, $(1,-,2)$ and $(1,3,2)$ match, and $(1,-,2)$ and $(1,3,-)$ also match. However, $(1,-,2)$ and $(0,-,2)$ do not match because their leading entries conflict. If τ and σ are both complete type vectors, then τ matches σ if and only if $\tau = \sigma$.

Definition 3.6 (Type constraints)**.** Let $\tau = (\tau_1, \ldots, \tau_n) \in \{0,1,2,3,-\}^n$ be a partial type vector. The *type constraints* for τ are a collection of equality constraints and non-strict inequality constraints on an arbitrary point $\mathbf{x} = (x_1, \ldots, x_{3n}) \in \mathbb{R}^{3n}$. For each $i = 1, \ldots, n$, the type symbol τ_i contributes the following constraints to this collection:

$x_{3i-2} = x_{3i-1} = x_{3i} = 0$	if $\tau_i = 0$;
$x_{3i-2} \geq 1$ and $x_{3i-1} = x_{3i} = 0$	if $\tau_i = 1$;
$x_{3i-1} \geq 1$ and $x_{3i-2} = x_{3i} = 0$	if $\tau_i = 2$;
$x_{3i} \geq 1$ and $x_{3i-2} = x_{3i-1} = 0$	if $\tau_i = 3$;
no constraints	if $\tau_i = -$.

Note that, unlike all of the other constraints seen so far, the type constraints are not invariant under scaling. The type constraints are most useful in the solution cone $\mathscr{Q}^\vee$, where any admissible point $\mathbf{x} \in \mathscr{Q}^\vee$ with $x_j > 0$ can be rescaled to some admissible point $\lambda\mathbf{x} \in \mathscr{Q}^\vee$ with $\lambda x_j \geq 1$. We see this scaling behaviour in Lemma 3.7 below.

The type constraints are similar but not identical to the conditions described by equation (3.1) for the type vector $\tau(\mathbf{x})$, and their precise relationship is described by the following lemma. The most important difference is that (3.1) uses strict inequalities of the form $x_j > 0$, whereas the type constraints use non-strict inequalities of the form $x_j \geq 1$. This is because the type constraints will be used with techniques from linear programming, where non-strict inequalities are simpler to deal with.

Lemma 3.7. *Let $\sigma \in \{0, 1, 2, 3, -\}^n$ be any partial type vector. If $\mathbf{x} \in \mathbb{R}^{3n}$ satisfies the type constraints for σ, then the type vector $\tau(\mathbf{x})$ matches σ. Conversely, if $\mathbf{x} \in \mathbb{R}^{3n}$ is an admissible vector whose type vector $\tau(\mathbf{x})$ matches σ, then $\lambda\mathbf{x}$ satisfies the type constraints for σ for some scalar $\lambda > 0$.*

Proof. This is a simple exercise in matching Definitions 3.1 and 3.6. Recall that admissible vectors $\mathbf{x}$ must satisfy $\mathbf{x} \geq \mathbf{0}$, which is why we are able to convert $x_j \neq 0$ into $\lambda x_j \geq 1$. □

We finish this section on type vectors with a simple but important note.

Lemma 3.8. *In the projective solution space, type vectors are always non-zero. That is, there is no $\mathbf{x} \in \mathscr{Q}$ for which $\tau(\mathbf{x}) = \mathbf{0}$.*

Proof. If $\tau(\mathbf{x}) = \mathbf{0}$ then $\mathbf{x} = \mathbf{0}$, by Definition 3.1. However, every point in the projective solution space lies in the hyperplane $\sum x_i = 1$. □

3.2 The structure of the algorithm

Recall that our overall plan is to iterate through all possible combinations of zero and non-zero coordinates. We do this by iterating through all possible type vectors, thereby implicitly enforcing the quadrilateral constraints as we go. The framework for this iteration is the *type tree*, which we define as follows.

Definition 3.9 (Type tree)**.** The *type tree* is a rooted tree of height n, where all leaf nodes are at depth n, and where each non-leaf node has precisely four children. The four edges descending from each non-leaf node are labelled 0, 1, 2 and 3 respectively.

Each node is labelled with a partial type vector. The root node is labelled $(-, \ldots, -)$. Each non-leaf node at depth i has a label of the form $(\tau_1, \ldots, \tau_i, -, -, \ldots, -)$, and its children along edges 0, 1, 2 and 3 have labels $(\tau_1, \ldots, \tau_i, 0, -, \ldots, -)$, $(\tau_1, \ldots, \tau_i, 1, -, \ldots, -)$, $(\tau_1, \ldots, \tau_i, 2, -, \ldots, -)$ and $(\tau_1, \ldots, \tau_i, 3, -, \ldots, -)$ respectively. Figure 3 illustrates the type tree of height $n = 2$.

The algorithm walks through the type tree in a depth-first manner, collecting admissible vertices of $\mathscr{Q}$ as it goes. The tree is large however, with $O(4^n)$ nodes, and so we do not traverse the type tree in full. Instead we prune the tree so that we only follow edges that might lead to admissible vertices. The main tools that we use for pruning are feasibility tests (based on the type constraints) and domination tests (based on the previous vertices found so far). The details are as follows.

Algorithm 3.10 (Tree traversal algorithm)**.** *The following algorithm takes a 3-manifold triangulation as input, and outputs the set of all admissible vertices of the projective solution space $\mathscr{Q}$.*

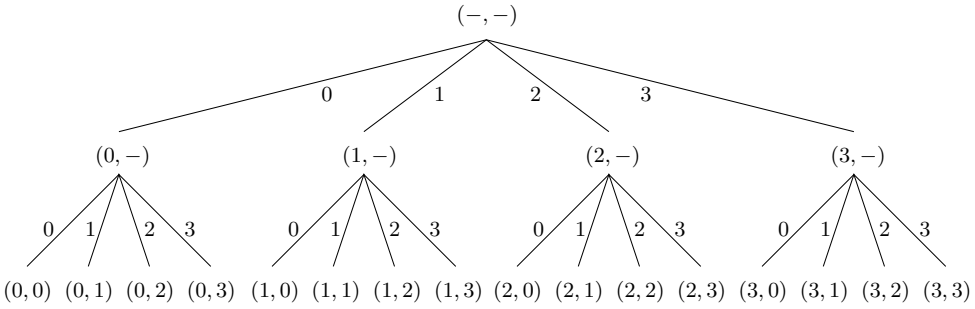

Figure 3: The type tree of height $n = 2$

Construct the matching equations M (see [9] for details), and delete rows until M has full rank. Initialise an empty set V; this will be used to store the type vectors for admissible vertices of $\mathscr{Q}$.

Beginning at the root of the type tree, process nodes recursively according to the following instructions. In general, we process the node N as follows:

- *If N is a non-leaf node then examine the four children of N in turn, beginning with the child along edge 0 (the order of the remaining children is unimportant). For a child node labelled with the partial type vector τ, recursively process this child if and only if all of the following conditions are satisfied:*
  - *(a)* Zero test: *τ is not the zero vector.*
  - *(b)* Domination test: *If we replace every unknown symbol $-$ with 0, then τ does not dominate any type vector in V.*
  - *(c)* Feasibility test: *There is some point $\mathbf{x} \in \mathbb{R}^{3n}$ that satisfies $\mathbf{x} \geq \mathbf{0}$, $M\mathbf{x} = \mathbf{0}$, and the type constraints for τ.*
- *If N is a leaf node then its label is a complete type vector τ, and we claim that τ must in fact be the type vector of an admissible vertex of $\mathscr{Q}$ (we prove this in Theorem 3.11). Insert τ into the set V, reconstruct the full vertex $\mathbf{x} \in \mathbb{R}^{3n}$ from τ as described by Lemma 3.2, and output this vertex.*

Both the domination test and the feasibility test are non-trivial. We discuss the details of these tests in Sections 3.3 and 3.4 below, along with the data structure used to store the solution set V. On the other hand, the zero test is simple; moreover, it is easy to see that it only needs to be tested once (for the very first leaf node that we process).

From the viewpoint of normal surface theory, it is important that this algorithm be implemented using *exact arithmetic*. This is because each admissible vertex $\mathbf{x} \in \mathscr{Q}$ needs to be scaled to its smallest integer multiple $\lambda\mathbf{x} \in \mathbb{Z}^{3n}$ in order to extract useful information for topological applications.

This is problematic because both the numerators and the denominators of the rationals that we encounter can grow exponentially large in n. For a naïve implementation we could simply use arbitrary-precision rational arithmetic (as provided for instance by the GMP library [21]). However, the bounds that we prove in Section 4 of this paper allow us to use native machine integer types (such as 32-bit, 64-bit or 128-bit integers) for many reasonable-sized applications.

Theorem 3.11. *Algorithm 3.10 is correct. That is, the output is precisely the set of all admissible vertices of $\mathscr{Q}$.*

Proof. By Lemma 3.2 the algorithm is correct if, when it terminates, the set V is precisely the set of type vectors of all admissible vertices of $\mathscr{Q}$. We prove this latter claim in three stages.

1. *Every type vector in V is the type vector of some admissible point $\mathbf{x} \in \mathscr{Q}$.*

   Let σ be some type vector in V. Because we reached the corresponding leaf node in our tree traversal, we know that σ passes the feasibility test. That is, there is some $\mathbf{x} \in \mathbb{R}^{3n}$ that satisfies $\mathbf{x} \geq 0$, $M\mathbf{x} = \mathbf{0}$, and the type constraints for σ. Moreover, σ also passes the zero test, which means we can rescale $\mathbf{x}$ so that $\sum x_i = 1$.

   Because σ was obtained from a leaf node it must be a complete type vector, which means that the type constraints for σ also enforce the quadrilateral constraints. In other words, this point $\mathbf{x}$ must be an admissible point in $\mathscr{Q}$.

   From Lemma 3.7 we know that $\tau(\mathbf{x})$ matches σ, and because σ is complete it follows that $\tau(\mathbf{x}) = \sigma$. That is, σ is the type vector of an admissible point in $\mathscr{Q}$.

2. *For every admissible vertex $\mathbf{x}$ of $\mathscr{Q}$, the type vector $\tau(\mathbf{x})$ appears in V.*

   All we need to show here is that we can descend from the root of the type tree to the leaf node labelled $\tau(\mathbf{x})$ without being stopped by the zero test, the domination test or the feasibility test.

   Let N be any ancestor of the leaf labelled $\tau(\mathbf{x})$ (possibly including the leaf itself), and let σ be its label. This means that σ can be obtained from $\tau(\mathbf{x})$ by replacing zero or more entries with the unknown symbol. In particular, σ matches $\tau(\mathbf{x})$.

   We know that σ passes the zero test by Lemma 3.8. From Lemma 3.7 there is some $\lambda > 0$ for which $\lambda\mathbf{x}$ satisfies the type constraints for σ. Since $\mathbf{x} \in \mathscr{Q}$ it is clear also that $\lambda\mathbf{x} \geq \mathbf{0}$ and $M\lambda\mathbf{x} = \mathbf{0}$, and so σ passes the feasibility test.

   By Lemma 3.4 the type vector $\tau(\mathbf{x})$ does not dominate $\tau(\mathbf{y})$ for any admissible point $\mathbf{y} \in \mathscr{Q}$, and from stage 1 above it follows that $\tau(\mathbf{x})$ does not dominate any type vector in V. The same is still true if we replace some elements of $\tau(\mathbf{x})$ with 0 (since this operation cannot introduce new dominations), which means that σ passes the domination test.

3. *Every type vector in V is the type vector of some admissible vertex $\mathbf{x} \in \mathscr{Q}$.*

   Suppose that some $\sigma \in V$ is not the type vector of an admissible vertex. By stage 1 above we know there is *some* admissible point $\mathbf{x} \in \mathscr{Q}$ for which $\sigma = \tau(\mathbf{x})$. Because $\mathbf{x}$ is not a vertex, it follows from Lemma 3.4 that $\sigma \geq \tau(\mathbf{y})$ for some admissible vertex $\mathbf{y} \in \mathscr{Q}$.

   By stage 2 above we know that $\tau(\mathbf{y})$ also appears in V. Since $\sigma \geq \tau(\mathbf{y})$, the type vector $\tau(\mathbf{y})$ can be obtained from σ by replacing one or more entries with 0. This means that the algorithm processes the leaf labelled $\tau(\mathbf{y})$ *before* the leaf labelled σ, yielding a contradiction: the node labelled σ should never have been processed, since it must have failed the domination test.

Steps 2 and 3 together show that V is precisely the set of type vectors of all admissible vertices of $\mathscr{Q}$, and the algorithm is therefore proven correct. □

Remark. All three tests (zero, domination and feasibility) can be performed in polynomial time in the input and/or output size. What prevents the entire algorithm from running in polynomial time is *dead ends*: nodes that we process but which do not eventually lead to any admissible vertices.

It is worth noting that Fukuda et al. [19] describe a backtracking framework that does not suffer from dead ends. However, this comes at a significant cost: before processing each node they must solve the *restricted vertex problem*, which essentially asks whether a vertex exists beneath a given node in the search tree. They prove this restricted vertex problem to be NP-complete, and they do not give an explicit algorithm to solve it.

We discuss time and space complexity further in Section 5, where we find that—despite the presence of dead ends—this tree traversal algorithm enjoys significantly lower worst-case complexity bounds than the prior state of the art. This is supported with experimental evidence in Section 6. In the meantime, it is worth noting some additional advantages of the tree traversal algorithm:

- *The tree traversal algorithm lends itself well to parallelisation.*

  The key observation here is that, for each non-leaf node N, the children along edges 1, 2 and 3 can be processed simultaneously (though edge 0 still needs to be processed first). There is no need for these three processes to communicate, since they cannot affect each other's domination tests. This three-way branching can be carried out repeatedly as we move deeper through the type tree, allowing us to make effective use of a large number of processors if they are available. Distributed processing is also practical, since at each branch the data transfer has small polynomial size.

- *The tree traversal algorithm supports incremental output.*

  If the type vectors of admissible vertices are reasonably well distributed across the type tree (as opposed to tightly clustered in a few sections of the tree), then we can expect a regular stream of output as the algorithm runs. From practical experience, this is indeed what we see: for problems that run for hours or even days, the algorithm outputs solutions at a continual (though slowing) pace, right from the beginning of the algorithm through until its termination.

  This incremental output is important for two reasons:

  – It allows *early termination* of the algorithm. For many topological decision algorithms, our task is to find *any* admissible vertex with some property P (or else conclude that no such vertex exists). As the tree traversal algorithm outputs vertices we can test them immediately for property P, and as soon as such a vertex is found we can terminate the algorithm.

  – It further assists with *parallelisation* for problems in which testing for property P is expensive (this is above and beyond the parallelisation techniques described above). An example is the Hakenness testing problem, where the test for property P can be far more difficult than the original normal surface enumeration [14]. For problems such as these, we can have a main tree traversal process that outputs vertices into a queue for processing, and separate dedicated processes that simultaneously work through this queue and test vertices for property P.

  This is a significant improvement over the double description method (the prior state of the art for normal surface enumeration). As outlined in Section 2, the double description method works inductively through a series of stages, and it typically does not output any solutions at all until it reaches the final stage.

- *The tree traversal algorithm supports progress tracking.*

  For any given node in the type tree, it is simple to estimate when it appears (as a percentage of running time) in a depth-first traversal of the full tree. We can present

these estimates to the user as the tree traversal algorithm runs, in order to give them some indication of the running time remaining.

Of course such estimates are inaccurate: they ignore pruning (which allows us to avoid significant portions of the type tree), and they ignore the variability in running times for the feasibility and domination tests (for instance, the domination test may slow down as the algorithm progresses and the solution set V grows). Nevertheless, they provide a rough indication of how far through the algorithm we are at any given time.

Again this is a significant improvement over the double description method. Some of the stages in the double description method can run many orders of magnitude slower than others [2, 9], and it is difficult to estimate in advance how slow each stage will be. In this sense, the obvious progress indicators (i.e., which stage we are up to and how far through it we are) are extremely poor indicators of global progress through the double description algorithm.

3.3 Implementing the domination test

What makes the domination test difficult is that the solution set V can grow exponentially large [10]. For closed triangulations, the best known theoretical bound is $|V| \in O\left(\left[\frac{3+\sqrt{13}}{2}\right]^n\right) \simeq O\left(3.303^n\right)$ as proven in [13], and for bounded triangulations there is only the trivial bound $|V| \leq 4^n$ (the total number of leaves in the type tree).

The central operation in the domination test is to decide, given a complete[3] type vector τ, whether there exists some $\sigma \in V$ for which $\tau \geq \sigma$. A naïve implementation simply walks through the entire set V, giving a time complexity of $O(n|V|)$ (since each individual test of $\tau \geq \sigma$ runs in $O(n)$ time).

However, we can do better with an efficient data structure for V. A key observation is that, although we might have up to 4^n type vectors in V, there are at most 2^n candidate type vectors σ that could possibly be dominated by τ. More precisely, suppose that τ contains k non-zero entries and $n-k$ zero entries (where $0 \leq k \leq n$). Then the only possible type vectors that τ could dominate are those obtained by replacing some of the non-zero entries in τ with zeroes, yielding 2^k such candidates in total.

We could therefore store V using a data structure that supports fast insertion and fast lookup (such as a red-black tree), and implement the domination test by enumerating all 2^k candidate vectors σ and searching for each of them in V. This has a time complexity of $O(n2^k \log |V|)$, which can be simplified to $O(n^2\, 2^k)$.

Although this looks better than the naïve implementation in theory, it can be significantly worse in practice. This is because, although $|V|$ has a theoretical upper bound of either $O(3.303^n)$ or 4^n, in practice it is far smaller. Detailed experimentation on closed triangulations [10] suggests that on average $|V|$ grows at a rate below 1.62^n, which means that enumerating all candidate vectors σ can be far worse than simply testing every $\sigma \in V$.

A compromise is possible that gives the best of both worlds. We can represent V using a tree structure that mirrors the type tree but stores only those nodes with descendants in V. This is essentially a trie (i.e., a prefix tree), and is illustrated in Figure 4.

To implement the domination test we simply walk through the portion of the trie that corresponds to nodes dominated by τ. This walk spans at most 2^k leaf nodes plus at most $n2^k$ ancestors. However, we also know that the trie contains at most $n|V|$ nodes in total (ancestors included). The walk therefore covers $O(\min\{n|V|, n2^k\})$ nodes, and since each comparison $\tau \geq \sigma$ comes for free with the structure of the trie, we have an overall running time of $O(\min\{n|V|, n2^k\})$ for the domination test.

[3] Recall that for the domination test we replace every unknown symbol $-$ with 0.

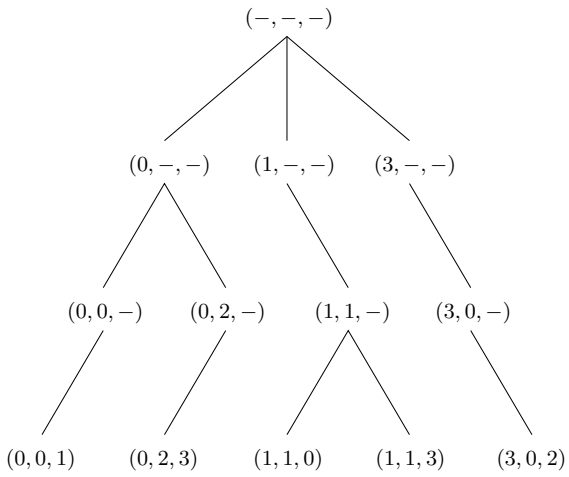

Figure 4: A trie representing $V = \{(0,0,1),\ (0,2,3),\ (1,1,0),\ (1,1,3),\ (3,0,2)\}$

3.4 Implementing the feasibility test

Recall that the feasibility test asks whether any point $\mathbf{x} \in \mathbb{R}^{3n}$ satisfies $\mathbf{x} \geq \mathbf{0}$, $M\mathbf{x} = \mathbf{0}$, and the type constraints for a given partial type vector τ. Tests of this kind are standard in the linear programming literature, and can be performed in polynomial time using interior point methods [17, 28].

However, our implementation of the feasibility test is based not on interior point methods, but on the *dual simplex method*. This is a pivot-based algorithm, and although it has a worst-case exponential running time in theory, it is simple to code and performs extremely well in practice [33, 36]. Moreover, the dual simplex method makes it easy to add new constraints to a previously-solved problem, allowing us to *incrementally* update our solutions $\mathbf{x} \in \mathbb{R}^{3n}$ as we walk through the type tree. For details on the dual simplex method, the reader is referred to a standard text on linear programming such as [5].

Here we outline the framework of the dual simplex method as it applies to the feasibility test. We explicitly describe the various matrices and vectors that appear, since these matrices and vectors play a key role in the results of Section 4.

Our first step is to simplify the integer matrix of matching equations M:

Definition 3.12 (Reduced matching equations)**.** Let M be the integer matrix of matching equations, as described in Section 2. The *reduced matching equation matrix* $\widetilde{M}$ is obtained from M by dividing each non-zero column by its greatest common divisor, which is always taken to be positive.

An example matrix M and the corresponding reduced matrix $\widetilde{M}$ are illustrated below. The first and fourth columns are divided by 2, and the remaining columns are left unchanged.

$$M = \begin{bmatrix} 0 & 1 & -1 & 2 & -1 & -1 \\ -2 & 0 & 2 & -2 & 0 & 2 \end{bmatrix} \implies \widetilde{M} = \begin{bmatrix} \mathbf{0} & 1 & -1 & \mathbf{1} & -1 & -1 \\ \mathbf{-1} & 0 & 2 & \mathbf{-1} & 0 & 2 \end{bmatrix}$$

It is clear from Lemma 2.2 that each greatest common divisor is 1, 2, 3 or 4, and in the case of an orientable triangulation just 1 or 2. Although this reduction is small, it turns out to have a significant impact on the arithmetical bounds that we prove in Section 4.

Although a solution to $M\mathbf{x} = \mathbf{0}$ need not satisfy $\widetilde{M}\mathbf{x} = \mathbf{0}$ (and vice versa), the two matrices are interchangeable for our purposes as shown by the following simple lemma.

Lemma 3.13. *Replacing M with $\widetilde{M}$ does not affect the results of the feasibility test. In other words, for any partial type vector τ the following two statements are equivalent:*

- *There exists an $\mathbf{x} \in \mathbb{R}^{3n}$ satisfying $\mathbf{x} \geq \mathbf{0}$, $M\mathbf{x} = \mathbf{0}$ and the type constraints for τ;*
- *There exists an $\mathbf{x}' \in \mathbb{R}^{3n}$ satisfying $\mathbf{x}' \geq \mathbf{0}$, $\widetilde{M}\mathbf{x}' = \mathbf{0}$ and the type constraints for τ.*

Proof. For each $i = 1, \ldots, 3n$, let d_i denote the greatest common divisor of the ith column of M (or 1 if the ith column of M is zero). In addition, let $D = \operatorname{lcm}(d_1, \ldots, d_{3n})$.

Suppose that $\mathbf{x} = (x_1, \ldots, x_{3n})$ satisfies $\mathbf{x} \geq \mathbf{0}$, $M\mathbf{x} = \mathbf{0}$ and the type constraints for τ. Then it is clear that $\mathbf{x}' = (d_1 x_1, \ldots, d_{3n} x_{3n})$ satisfies $\mathbf{x}' \geq \mathbf{0}$, $\widetilde{M}\mathbf{x}' = \mathbf{0}$ and the type constraints for τ.

Conversely, suppose that $\mathbf{x}' = (x'_1, \ldots, x'_{3n})$ satisfies $\mathbf{x}' \geq \mathbf{0}$, $\widetilde{M}\mathbf{x}' = \mathbf{0}$ and the type constraints for τ. In this case, we find that $\mathbf{x} = (Dx'_1/d_1, \ldots, Dx'_{3n}/d_{3n})$ satisfies $\mathbf{x} \geq \mathbf{0}$, $M\mathbf{x} = \mathbf{0}$ and the type constraints for τ. □

Our next step is to replace type constraints of the form $x_i \geq 1$ with equivalent constraints of the form $x'_i \geq 0$.

Lemma 3.14. *Let τ be any partial type vector, and let the type constraints for τ be $x_i = 0$ for all $i \in I$ and $x_j \geq 1$ for all $j \in J$. Then the following two statements are equivalent:*

- *There exists an $\mathbf{x} \in \mathbb{R}^{3n}$ for which $\mathbf{x} \geq \mathbf{0}$, $\widetilde{M}\mathbf{x} = \mathbf{0}$, $x_i = 0$ for all $i \in I$ and $x_j \geq 1$ for all $j \in J$;*
- *There exists an $\mathbf{x}' \in \mathbb{R}^{3n}$ for which $\mathbf{x}' \geq \mathbf{0}$, $\widetilde{M}\mathbf{x}' = \mathbf{b}$ and $x'_i = 0$ for all $i \in I$, where $\mathbf{b} = -\sum_{j \in J} \widetilde{M}_j$, and where $\widetilde{M}_j$ refers to the jth column of $\widetilde{M}$.*

Proof. The two statements are equivalent under the substitution $x'_j = x_j - 1$ for $j \in J$ and $x'_j = x_j$ for $j \notin J$. □

Based on Lemmata 3.13 and 3.14, we structure the feasibility test as follows.

Algorithm 3.15 (Framework for feasibility test)**.** *To perform the feasibility test, we use the dual simplex method to search for a solution $\mathbf{x} \in \mathbb{R}^{3n}$ for which $\mathbf{x} \geq \mathbf{0}$, $\widetilde{M}\mathbf{x} = \mathbf{b}$ and $x_i = 0$ for all $i \in I$. Here the vector $\mathbf{b}$ and the index set I reflect the partial type vector τ that we are working with, and we construct both $\mathbf{b}$ and I incrementally as follows:*

- *When we begin at the root node with $\tau = (-, \ldots, -)$, we set $\mathbf{b} = \mathbf{0}$ and $I = \emptyset$.*
- *Suppose we step down from a parent node labelled $(\tau_1, \ldots, \tau_k, -, -, \ldots, -)$ to a child node labelled $(\tau_1, \ldots, \tau_k, \tau_{k+1}, -, \ldots, -)$.*
  - *If $\tau_{k+1} = 0$ then we insert the three new indices $3k+1$, $3k+2$ and $3k+3$ into the set I, reflecting the additional type constraints $x_{3k+1} = x_{3k+2} = x_{3k+3} = 0$.*
  - *If $\tau_{k+1} = 1$ then we insert the two new indices $3k+2$ and $3k+3$ into I and subtract the column $\widetilde{M}_{3k+1}$ from $\mathbf{b}$, reflecting the additional type constraints $x_{3k+2} = x_{3k+3} = 0$ and $x_{3k+1} \geq 1$.*
  - *The cases $\tau_{k+1} = 2$ and 3 are handled in a similar fashion to $\tau_{k+1} = 1$ above.*

It is important to note that this "step down" operation involves only minor changes to $\mathbf{b}$ and I. This means that, instead of solving each child feasibility problem from scratch, we can use the solution $\mathbf{x}$ from the parent node as a starting point for the new dual simplex search when we introduce our new constraints. We return to this shortly.

Recall from Algorithm 3.10 that M is full rank, and suppose that $\operatorname{rank} M = \operatorname{rank} \widetilde{M} = k$. To perform the feasibility test we maintain the following structures (all of which are common to revised simplex algorithms, and again we refer the reader to a standard text [5] for details):

- a *basis* $\beta = \{\beta_1, \ldots, \beta_k\} \subseteq \{1, \ldots, 3n\}$, which identifies k linearly independent columns of $\widetilde{M}$;
- the $k \times k$ inverse matrix $\widetilde{M}_\beta^{-1}$, where $\widetilde{M}_\beta$ denotes the $k \times k$ matrix formed from columns $\beta_1, \ldots, \beta_k$ of $\widetilde{M}$.

It is useful to think of $\widetilde{M}_\beta^{-1}$ as a matrix of row operations that we can apply to both $\widetilde{M}$ and $\mathbf{b}$. The product $\widetilde{M}_\beta^{-1}\widetilde{M}$ includes the $k \times k$ identity as a submatrix, and the product $\widetilde{M}_\beta^{-1}\mathbf{b}$ identifies the elements of a solution vector $\mathbf{x}$ as follows.

Any basis β describes a solution $\mathbf{x} \in \mathbb{R}^{3n}$ to the equation $\widetilde{M}\mathbf{x} = \mathbf{b}$, in which $x_j = 0$ for each $j \notin \beta$, and where the product $\widetilde{M}_\beta^{-1}\mathbf{b}$ lists the remaining elements $(x_{\beta_1}, \ldots, x_{\beta_k})$. Such a solution only satisfies $\mathbf{x} \geq \mathbf{0}$ if we have $\widetilde{M}_\beta^{-1}\mathbf{b} \geq \mathbf{0}$. We call the basis *feasible* if $\widetilde{M}_\beta^{-1}\mathbf{b} \geq \mathbf{0}$, and we call it *infeasible* if $\widetilde{M}_\beta^{-1}\mathbf{b} \ngeq \mathbf{0}$.

The aim of the dual simplex method is to find a feasible basis. It does this by constructing some initial (possibly infeasible) basis,[4] and then repeatedly modifying this basis using *pivots*. Each pivot replaces some index in β with some other index not in β, and corresponds to a simple sequence of row operations that we apply to $\widetilde{M}_\beta^{-1}$. Eventually the dual simplex method produces either a feasible basis or a certificate that no feasible basis exists. We apply this procedure to our problem as follows:

Algorithm 3.16 (Details of feasibility test)**.** *At the root node of the type tree, we simply construct any initial basis and then pivot until we obtain either a feasible basis or a certificate that no feasible basis exists (in which case the root node fails the feasibility test).*

When we step down from a parent node to a child node, we "inherit" the feasible basis from the parent and use this as the initial basis for the child. This may no longer be feasible because $\mathbf{b}$ *might have changed; if not then once again we pivot until the basis can be made feasible or we can prove that no feasible basis exists.*

Recall that we enlarge the index set I when we step to a child node, which means that we introduce new constraints of the form $x_j = 0$. Each constraint $x_j = 0$ is enforced as follows:

- *If j is not in the basis then we simply eliminate the variable x_j from the system (effectively deleting the corresponding column from $\widetilde{M}$).*
- *If j is in the basis then we pivot to remove it and eliminate x_j as before. If we cannot pivot j out of the basis then some row of the product $\widetilde{M}_\beta^{-1}\widetilde{M}$ must be zero everywhere except for column j, which means the equation $\widetilde{M}_\beta^{-1}\widetilde{M}\mathbf{x} = \widetilde{M}_\beta^{-1}\mathbf{b}$ implies $x_j = z$ for some scalar z. If $z \neq 0$ then it is impossible to enforce $x_j = 0$. If $z = 0$ then $x_j = 0$ is enforced automatically, and there is nothing we need to do.*

The child node is deemed to pass the feasibility test if and only if we are able to obtain a feasible basis after enforcing all of our new constraints.

The procedures described in Algorithm 3.16 are all standard elements of the dual simplex method, and once more we refer the reader to a standard text [5] for details and proofs. There are typically many choices available for which pivots to perform; one must be careful to choose a pivoting rule that avoids cycling, since the solution cone $\mathscr{Q}^\vee$ is highly degenerate. We use Bland's rule [7] in our implementation.

[4] The dual simplex method usually has an extra requirement that this initial basis be *dual feasible*. For our application there is no objective function to optimise, and so this extra requirement can be ignored.

4 Arithmetical bounds

The tree traversal algorithm involves a significant amount of rational arithmetic—most notably, in the feasibility test (Section 3.4) and the reconstruction of vertices of $\mathscr{Q}$ from their type vectors (Lemma 3.2). Moreover, the rationals that we work with can have exponentially large numerators and denominators, which means that in general we must use arbitrary precision arithmetic (as provided for instance by the GMP library [21]).

The aim of this section is to derive explicit bounds on the rational numbers that appear in our calculations. In particular, Theorem 4.4 bounds the elements of the matrix $\widetilde{M}_\beta^{-1}$ that we manipulate in the feasibility test, and Corollaries 4.5 and 4.6 bound the coordinates of the final vertices of $\mathscr{Q}$.

These results are useful because the relevant bounds can be precomputed at the beginning of the algorithm, and for many reasonable-sized problems they allow us to work in native machine integer types (such as 32-bit, 64-bit or 128-bit integers). These native types are significantly faster than arbitrary precision arithmetic: experimentation with the tree traversal algorithm shows a speed increase of over 10 times.

Furthermore, Corollary 4.6 is a significant improvement on the best known bounds for the coordinates of vertices of $\mathscr{Q}$. These have been independently studied by Hass, Lagarias and Pippenger [24] and subsequently by Burton, Jaco, Letscher and Rubinstein; we recount these earlier results before presenting Corollary 4.6 below.

Central to all of the results in this section is the *bounding constant* $\delta(\widetilde{M})$. This is a function of the reduced matching equation matrix $\widetilde{M}$, and to obtain the tightest possible bounds it can be computed at runtime when the tree traversal algorithm begins. If global quantities are required then Lemma 4.2 and Corollary 4.3 bound $\delta(\widetilde{M})$ in terms of n alone.

Definition 4.1 (Bounding constant)**.** The *bounding constant* $\delta(\widetilde{M})$ is defined as follows. For each column $c = 1, \ldots, 3n$, compute the Euclidean length of the cth column of $\widetilde{M}$; that is, $(\sum_i \widetilde{M}_{i,c}^2)^{\frac{1}{2}}$. We define $\delta(\widetilde{M})$ to be the product of the k largest of these lengths, where $k = \operatorname{rank} \widetilde{M}$.

For example, consider again the matrix

$$\widetilde{M} = \begin{bmatrix} 0 & 1 & -1 & 1 & -1 & -1 \\ -1 & 0 & 2 & -1 & 0 & 2 \end{bmatrix},$$

which has rank $k = 2$ and column lengths 1, 1, $\sqrt{5}$, $\sqrt{2}$, 1 and $\sqrt{5}$. Here the two largest column lengths are both $\sqrt{5}$, and so $\delta(\widetilde{M}) = \sqrt{5} \cdot \sqrt{5} = 5$.

Lemma 4.2. *For an arbitrary 3-manifold triangulation,* $\delta(\widetilde{M}) \leq (\sqrt{10})^{\operatorname{rank} \widetilde{M}} \leq (\sqrt{10})^{3n}$. *For an orientable triangulation, we can improve this bound to* $\delta(\widetilde{M}) \leq (\sqrt{6})^{\operatorname{rank} \widetilde{M}} \leq (\sqrt{6})^{3n}$.

Proof. Using Lemma 2.2 and the fact that each column of $\widetilde{M}$ has gcd = 1, there are only a few possibilities for the non-zero entries in a column of $\widetilde{M}$. In the orientable case the only options are $(\pm1, \pm1, \pm1, \pm1)$, $(\pm1, \pm1, \pm1)$, $(\pm1, \pm1)$, (±1), $(\pm2, \pm1, \pm1)$ and $(\pm2, \pm1)$. For non-orientable triangulations we have the additional possibility of $(\pm3, \pm1)$.

It follows that each column has Euclidean length $\leq \sqrt{6}$ in the orientable case and $\leq \sqrt{10}$ otherwise, yielding bounds of $\delta(\widetilde{M}) \leq (\sqrt{6})^{\operatorname{rank} \widetilde{M}}$ and $\delta(\widetilde{M}) \leq (\sqrt{10})^{\operatorname{rank} \widetilde{M}}$ respectively. To finish we note that $\widetilde{M}$ has $\leq 3n$ columns[5] and so $\operatorname{rank} \widetilde{M} \leq 3n$. □

A result of Tillmann [38] shows that $\operatorname{rank} M = n$ for any *closed* 3-manifold triangulation, which allows us to tighten our bounds further:

[5]When we begin the tree traversal algorithm the matrix $\widetilde{M}$ has *precisely* $3n$ columns, but recall from Section 3.4 that we might delete columns from $\widetilde{M}$ as we move through the type tree.

Corollary 4.3. *For an arbitrary closed 3-manifold triangulation we have* $\delta(\widetilde{M}) \leq (\sqrt{10})^n$, *and for a closed orientable 3-manifold triangulation we have* $\delta(\widetilde{M}) \leq (\sqrt{6})^n$.

Remark. In fact these bounds are general: both $\delta(\widetilde{M}) \leq (\sqrt{6})^n$ for the orientable case and $\delta(\widetilde{M}) \leq (\sqrt{10})^n$ otherwise hold for triangulations with boundary also. The argument relies on a proof that $\operatorname{rank} M = e - v \leq n$, where e and v are the number of edges and vertices of the triangulation that do not lie within the boundary. This proof involves topological techniques beyond the scope of this paper; see [11] for the full details.

We can now use $\delta(\widetilde{M})$ to bound the rational numbers that appear as we perform the feasibility test (Algorithms 3.15 and 3.16). Our main tool is *Hadamard's inequality*, which states that if A is any square matrix, $|\det A|$ is bounded above by the product of the Euclidean lengths of the columns of A.

Theorem 4.4. *Let* $\beta \subseteq \{1, \ldots, 3n\}$ *be any basis as described in Section 3.4, and let* $\widetilde{M}_\beta^{-1}$ *be the corresponding* $k \times k$ *inverse matrix, where* $k = \operatorname{rank} \widetilde{M}$. *Then* $\widetilde{M}_\beta^{-1}$ *can be expressed in the form* $\widetilde{M}_\beta^{-1} = N/\Delta$, *where* Δ *is an integer with* $|\Delta| \leq \delta(\widetilde{M})$, *and where* N *is an integer matrix with* $|n_{i,j}| \leq \delta(\widetilde{M})$ *for every* i, j.

Before proving this result, it is important to note that it holds for *any* basis—not just initial bases or feasible bases, but every basis that we pivot through along the way. This means that we can use Theorem 4.4 to place an upper bound on every integer that appears in every intermediate calculation, enabling us to use native machine integer types instead of arbitrary precision arithmetic when $\delta(\widetilde{M})$ is of a reasonable size.

For example, consider the Weber-Seifert space, which is mentioned in the introduction and has been considered one of the benchmarks for progress in computational topology. Using the 23-tetrahedron triangulation described in [14] we find that $\delta(\widetilde{M}) = 2^{23}$, which allows us to store the matrix $\widetilde{M}_\beta^{-1}$ using native (and very fast) 32-bit integers.[6]

Proof of Theorem 4.4. Using the adjoint formula for a matrix inverse we have $\widetilde{M}_\beta^{-1} = N/\Delta$, where $N = \operatorname{adj} \widetilde{M}_\beta$ and $\Delta = \det \widetilde{M}_\beta$. Because $\widetilde{M}$ is an integer matrix it is clear that Δ and all of the entries $n_{i,j}$ are integers.

By Hadamard's inequality, $|\Delta|$ is bounded above by the product of the Euclidean lengths of the k columns of $\widetilde{M}_\beta$, and it follows from Definition 4.1 that $|\Delta| \leq \delta(\widetilde{M})$. Since each adjoint entry $n_{i,j}$ is a subdeterminant of $\widetilde{M}_\beta$, a similar argument shows that $|n_{i,j}| \leq \delta(\widetilde{M})$ for each i, j. □

To finish this section, we use Theorem 4.4 to bound the coordinates of any admissible vertex $\mathbf{v} \in \mathscr{Q}$. More precisely, we bound the coordinates of $\mathbf{u} = \lambda \mathbf{v}$, where $\mathbf{u} \in \mathscr{Q}^\vee$ is the smallest *integer* multiple of $\mathbf{v}$. This smallest integer multiple has a precise meaning for topologists (essentially, each integer u_i corresponds to a collection of u_i quadrilaterals embedded in some tetrahedron of the triangulation; see [9, 39] for full details).

The first such bound was due to Hass, Lagarias and Pippenger [24], who proved that $|u_i| \leq 128^n/2$. Their result applies only to true simplicial complexes (not the more general triangulations that we allow here), and under their assumptions the matching equation matrix M is much simpler (containing only 0 and ± 1 entries).

The only other bound known to date appears in unpublished notes by Burton, Jaco, Letscher and Rubinstein (c. 2001 and referenced in [26]), where it is shown that $|u_i| \leq (\sqrt{8})^n$

[6] Of course we must be careful: for instance, row operations of the form $\mathbf{x} \leftarrow \lambda \mathbf{x} + \mu \mathbf{y}$ must be performed using 64-bit arithmetic, even though the inputs and outputs are guaranteed to fit into 32-bit integers.

for a one-vertex closed orientable triangulation (this time generalised triangulations are allowed).[7]

In Corollary 4.5 we obtain new bounds on $|u_i|$ in terms of the bounding constant $\delta(\widetilde{M})$, and in Corollary 4.6 we use this to bound $|u_i|$ in terms of n alone, yielding significant improvements to both the Hass et al. and Burton et al. bounds outlined above.

Corollary 4.5. *Let* $\mathbf{u}$ *be the smallest integer multiple of an admissible vertex* $\mathbf{v} \in \mathscr{Q}$*. Then each coordinate* u_i *is bounded as follows:*

- $|u_i| \leq (4nk + 2) \cdot \delta(\widetilde{M})$ *if the triangulation is orientable;*
- $|u_i| \leq (36nk + 12) \cdot \delta(\widetilde{M})$ *otherwise,*

where $k = \operatorname{rank} M = \operatorname{rank} \widetilde{M}$*.*

Proof. This proof is a simple (though slightly messy) matter of starting with Theorem 4.4 and following back through the various transformations and changes of variable until we reach our solution $\mathbf{v} \in \mathscr{Q}$ to the original matching equations.

Let τ be the type vector of the vertex $\mathbf{v}$, and let $\mathbf{x} \in \mathbb{R}^{3n}$ be the corresponding solution to the feasibility test as found by the dual simplex method in Algorithm 3.15 (so that $\mathbf{x} \geq \mathbf{0}$ and $\widetilde{M}\mathbf{x} = \mathbf{b}$). Let β be the corresponding feasible basis found by the dual simplex method, so that every non-zero entry in $\mathbf{x}$ appears as an entry of the vector $\widetilde{M}_\beta^{-1}\mathbf{b}$.

From Lemma 3.14 we recall that $\mathbf{b}$ is the negative sum of at most n columns of $\widetilde{M}$. Following the same argument used in the proof of Lemma 4.2, each non-zero entry in $\widetilde{M}$ is either ± 1, ± 2 or ± 3, and for an orientable triangulation just ± 1 or ± 2. It follows that each $|b_i| \leq 2n$ if our triangulation is orientable, and each $|b_i| \leq 3n$ otherwise.

We return now to the vector $\mathbf{x}$. As in Theorem 4.4, let $\widetilde{M}_\beta^{-1} = N/\Delta$, where Δ is an integer with $|\Delta| \leq \delta(\widetilde{M})$, and where each $n_{i,j}$ is an integer with $|n_{i,j}| \leq \delta(\widetilde{M})$. Every non-zero element of $\mathbf{x}$ appears in the vector $\widetilde{M}_\beta^{-1}\mathbf{b}$, and is therefore of the form $\sum_{j=1}^{k} n_{i,j} b_j / \Delta$. Combining all of the bounds above, we deduce that $\mathbf{x} = \mathbf{x}'/\Delta$, where $\mathbf{x}'$ is an integer vector and where each $|x'_i| \leq 2nk\delta(\widetilde{M})$ if our triangulation is orientable, or $|x'_i| \leq 3nk\delta(\widetilde{M})$ otherwise.

Our next task is to step backwards through Lemma 3.14. Our current vector $\mathbf{x}$ satisfies the second statement of Lemma 3.14; let $\mathbf{y}$ be the corresponding solution for the first statement, so that $\mathbf{y} \geq \mathbf{0}$, $\widetilde{M}\mathbf{y} = \mathbf{0}$, and $\mathbf{y}$ satisfies the type constraints for τ. As in the proof of Lemma 3.14, each $y_i = x_i$ or $x_i + 1$. It follows that $\mathbf{y} = \mathbf{y}'/\Delta$, where $\mathbf{y}'$ is an integer vector and where each $|y'_i| \leq 2nk\delta(\widetilde{M}) + \Delta$ if our triangulation is orientable, or $|y'_i| \leq 3nk\delta(\widetilde{M}) + \Delta$ otherwise.

Finally we step backwards through Lemma 3.13. Our vector $\mathbf{y}$ satisfies the second statement of Lemma 3.13; let $\mathbf{z}$ be the corresponding solution for the first statement, so that $\mathbf{z} \geq \mathbf{0}$, $M\mathbf{z} = \mathbf{0}$, and $\mathbf{z}$ satisfies the type constraints for τ. Following the proof of Lemma 3.13, each $z_i = Dy_i/d_i$, where d_i is the gcd of the ith column of M, and where $D = \operatorname{lcm}(d_1, \ldots, d_{3n})$. By Lemma 2.2 we have $D \leq 2$ for an orientable triangulation and $D \leq 12$ otherwise. Therefore $\mathbf{z} = \mathbf{z}'/\Delta$, where $\mathbf{z}'$ is an integer vector and where each $|z'_i| \leq 4nk\delta(\widetilde{M}) + 2\Delta$ if our triangulation is orientable, or $|z'_i| \leq 36nk\delta(\widetilde{M}) + 12\Delta$ otherwise.

To conclude, we observe that our vertex $\mathbf{v}$ is a multiple of $\mathbf{z}$, and so the entries in the smallest integer multiple $\mathbf{u}$ are bounded by the entries in the (possibly larger) integer multiple $\mathbf{z}'$. In particular, $|u_i| \leq 4nk\delta(\widetilde{M}) + 2\Delta \leq (4nk + 2) \cdot \delta(\widetilde{M})$ if our triangulation is orientable, or $|u_i| \leq 36nk\delta(\widetilde{M}) + 12\Delta \leq (36nk + 12) \cdot \delta(\widetilde{M})$ otherwise. □

[7] Both of these previously-known bounds were derived in standard coordinates ($\mathbb{R}^{7n}$), not quadrilateral coordinates ($\mathbb{R}^{3n}$). However, it is simple to convert between coordinate systems [9], and it can be shown that the upper bounds differ by a factor of at most $4n$.

We can recast these bounds purely in terms of n: for $\delta(\widetilde{M})$ we use Lemma 4.2, and for the rank k we use Tillmann's theorem that $\operatorname{rank} M = n$ for closed triangulations [38], or the observation that $\operatorname{rank} M \leq 3n$ for bounded triangulations. The result is the following.

Corollary 4.6. *Let* $\mathbf{u}$ *be the smallest integer multiple of an admissible vertex of* $\mathscr{Q}$*. Then each coordinate* u_i *is bounded as follows:*

- $|u_i| \leq (4n^2 + 2) \cdot (\sqrt{6})^n$ *if the triangulation is closed and orientable;*
- $|u_i| \leq (36n^2 + 12) \cdot (\sqrt{10})^n$ *if the triangulation is closed and non-orientable;*
- $|u_i| \leq (12n^2 + 2) \cdot (\sqrt{6})^{3n}$ *if the triangulation is bounded and orientable;*
- $|u_i| \leq (108n^2 + 12) \cdot (\sqrt{10})^{3n}$ *if the triangulation is bounded and non-orientable.*

Remark. These bounds are significant improvements over previous results. For bounded triangulations, we improve the Hass et al. bound of $O(128^n)$ to $O[n^2(\sqrt{10})^{3n}] \simeq O(n^2\, 31.6^n)$, whilst removing the requirement for a true simplicial complex. For closed orientable triangulations, we improve the Burton et al. bound of $O[(\sqrt{8})^n]$ to $O[n^2(\sqrt{6})^n]$, whilst removing the requirement for a one-vertex triangulation.

As with Corollary 4.3, the stronger bounds that we obtain for closed triangulations can in fact be shown to hold for bounded triangulations also. Again this relies on a proof that $\operatorname{rank} M = e - v \leq n$, which calls on techniques beyond the scope of this paper. See [11] for details.

5 Time and space complexity

In this section we derive theoretical bounds on the time and space complexity of normal surface enumeration algorithms. We compare these bounds for the double description method (the prior state of the art, as described in [12]) and the tree traversal algorithm that we introduce in this paper.

All of these bounds are driven by exponential factors, and it is these exponential factors that we focus on here. We replace any polynomial factors with a generic polynomial $\varphi(n)$; for instance, a complexity bound of $O(4^n n^3)$ would simply be reported as $O(4^n \varphi(n))$ here. For both algorithms it is easy to show that these polynomial factors are of small degree.

We assume that all arithmetical operations can be performed in polynomial time. For the tree traversal algorithm this is a simple consequence of Theorem 4.4, and for the double description method it can likewise be shown that all integers have $O(n)$ bits. We omit the details here.

Theorem 5.1 analyses the double description method, and Theorem 5.2 studies the tree traversal algorithm. In summary, the double description method can be performed in $O(16^n \varphi(n))$ time and $O(4^n \varphi(n))$ space, whereas the tree traversal algorithm can be carried out in $O(4^n |V| \varphi(n))$ time and $O(|V| \varphi(n))$ space, where $|V|$ is the output size. This is a significant reduction, given that the output size for normal surface enumeration is often extremely small in practice.[8] If we reformulate these bounds in terms of n alone, the tree traversal algorithm requires $O(7^n \varphi(n))$ time, and approximately $O(3.303^n \varphi(n))$ space for closed triangulations or $O(4^n \varphi(n))$ space for bounded triangulations.

We do not give bounds in terms of $|V|$ for the double description method, because it suffers from a well-known *combinatorial explosion*: even when the output size is small, the intermediate structures that appear can be significantly (and exponentially) more complex.

[8] Early indications of this appear in [10] (which works in the larger space $\mathbb{R}^{7n}$), and a detailed study will appear in [11]. To illustrate how extreme these results are in $\mathbb{R}^{3n}$: across all 139 103 032 closed 3-manifold triangulations of size $n = 9$, the maximum output size is just 37, and the *mean* output size is a mere 9.7.

See [2] for examples of this in theory, or [9] for experiments that show this in practice for normal surface enumeration. The result is that, even for bounded triangulations where both algorithms have space complexities of $O(4^n\varphi(n))$, the tree traversal algorithm can effectively exploit a small output size whereas the double description method cannot.

In conclusion, both the time and space bounds for tree traversal are notably smaller, particularly when the output size is small. However, it is important to bear in mind that for both algorithms these bounds may still be far above the "real" time and memory usage that we observe in practice. For this reason it is important to carry out real practical experiments, which we do in Section 6.

As a final note, for this theoretical analysis we always assume implementations that give the best possible complexity bounds. For the tree traversal algorithm we assume interior point methods for the feasibility test. For the double description method we assume that adjacency of vertices is tested using a polynomial-time algebraic rank test, rather than a simpler combinatorial test that has worst-case exponential time but excellent performance in practice [12, 20]. For the experiments of Section 6 we revert to the dual simplex method and combinatorial test respectively, both of which are known to perform extremely well in practical settings [12, 20, 33, 36].

Theorem 5.1. *The double description method for normal surface enumeration can be implemented in worst-case $O(16^n\varphi(n))$ time and $O(4^n\varphi(n))$ space.*

Proof. As outlined in Section 2, the double description method constructs a series of polytopes $P_0, \ldots, P_e \subseteq \mathbb{R}^{3n}$, where each P_i is the intersection of the unit simplex with the first i matching equations. There are $e+1 \in O(n)$ such polytopes in total, and with an algebraic adjacency test we can construct each P_i in $O(v_{i-1}^2\varphi(n))$ time and $O(v_{i-1}\varphi(n))$ space, where v_{i-1} is the number of vertices of P_{i-1} that satisfy the quadrilateral constraints [12].

The main task then is to estimate each v_i. In every polytope P_i, each vertex is uniquely determined by which of its coordinates are zero [12]. There are 4^n possible choices of zero coordinates that satisfy the quadrilateral constraints, and so each $v_i \leq 4^n$. This gives a time complexity of $O(16^n\varphi(n))$ and space complexity of $O(4^n\varphi(n))$ for the algorithm overall. □

It should be noted that there are other ways of estimating the vertex counts v_i. For the final polytope, Burton [13] proves that $v_e \in O(3.303^n)$ for a closed triangulation; however, this result does not hold for the intermediate vertex counts $v_1, \ldots, v_{e-1}$. McMullen's theorem [32] gives a tight bound on the number of vertices of an arbitrary polytope, but here it only yields $v_i \in O(4.24^n)$; the reason it exceeds 4^n is that it does not take into account the quadrilateral constraints.

Theorem 5.2. *The tree traversal algorithm for normal surface enumeration can be implemented in worst-case $O(4^n|V|\varphi(n))$ time and $O(|V|\varphi(n))$ space, where $|V|$ is the number of admissible vertices of $\mathscr{Q}$. In terms of n alone, the worst-case time complexity is $O(7^n\varphi(n))$, and the worst-case space complexity is $O\left(\left[\frac{3+\sqrt{13}}{2}\right]^n\varphi(n)\right) \simeq O(3.303^n\varphi(n))$ for closed triangulations or $O(4^n\varphi(n))$ for bounded triangulations.*

Proof. The space complexity is straightforward. We do not need to explicitly construct the type tree in memory in order to traverse it, and so the only non-polynomial space requirements come from storing the solution set V. It follows that the tree traversal algorithm has space complexity $O(|V|\varphi(n))$. By the same argument as before we have $|V| \leq 4^n$ for arbitrary triangulations, and for closed triangulations a result of Burton [13] shows that $|V| \in O\left(\left[\frac{3+\sqrt{13}}{2}\right]^n\right)$.

The time complexity is more interesting. There are $O(4^n)$ nodes in the type tree, and the only non-polynomial operation that we perform on each node is the domination test. As

described in Section 3.3, for a node labelled with the partial type vector τ the domination test takes $O(\min\{n|V|,\ n2^k\})$ time, where k is the number of times the symbols 1, 2 or 3 appear in τ. It follows that the tree traversal algorithm runs in $O(4^n|V|\varphi(n))$ time.

To obtain a complexity bound that does not involve $|V|$, a simple counting argument shows that at most $n\binom{n}{k}3^k$ nodes in the type tree are labelled with a partial type vector τ with the property described above. Therefore the total running time is bounded by

$$O\left(\left[\sum_{k=0}^{n} n\binom{n}{k}3^k \cdot n2^k\right]\varphi(n)\right) = O\left(\left[\sum_{k=0}^{n}\binom{n}{k}6^k\right]\varphi(n)\right) = O\left(7^n\varphi(n)\right),$$

using the binomial expansion $\sum_{k=0}^{n}\binom{n}{k}6^k = (1+6)^n = 7^n$. □

6 Experimental performance

In this section we put our algorithm into practice. We run it through a test suite of 275 triangulations, based on the first 275 entries in the census of knot complements tabulated by Christy and shipped with version 1.9 of *Snap* [15].

All 275 of these triangulations are *bounded* triangulations. We use bounded triangulations because they are a stronger "stress test": in practice it has been found that enumerating normal surfaces for a bounded triangulation is often significantly more difficult than for a closed triangulation of comparable size [14]. In part this is because the number of admissible vertices of $\mathscr{Q}$ is typically much larger in the bounded case [11].

We begin this section by studying the growth of the number of admissible vertices and the number of "dead ends" that we walk through in the type tree, where we discover that the number of dead ends is far below the theoretical bound of $O(4^n)$. We follow with a direct comparison of running times for the tree traversal algorithm and the double description method, where we find that our new algorithm runs slower for smaller problems but significantly faster for larger and more difficult problems.

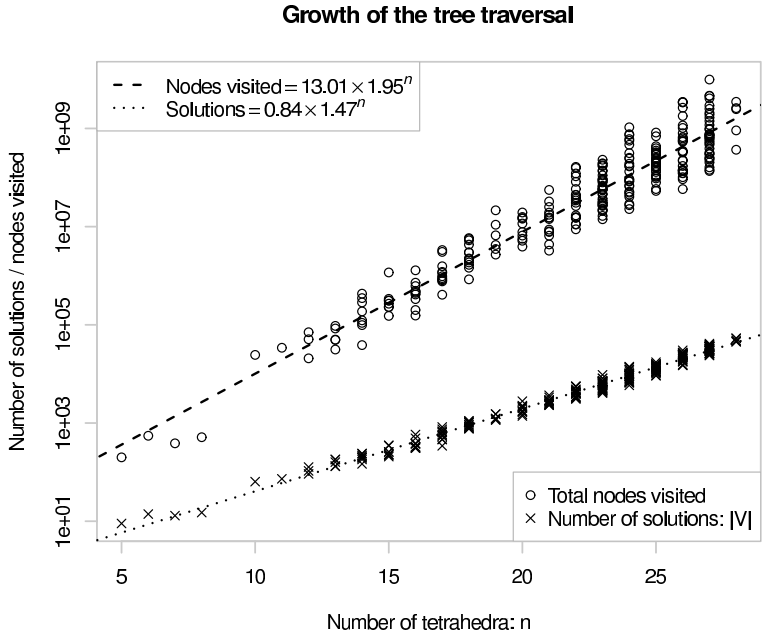

Figure 5: Counting solutions and dead ends for the tree traversal algorithm

Figure 5 measures the "combinatorial size" of the tree traversal: the crosses in the lower part of the graph represent the final number of solutions, and the circles in the upper part of the graph count the total number of nodes that we visit (including solutions as well as dead ends). In the theoretical analysis of Section 5 we estimate both figures as $O(4^n)$, but in practice we find that the real counts are significantly smaller. The number of solutions appears to grow at a rate of roughly 1.47^n, and the number of nodes that we visit grows at a rate of roughly 1.95^n (though with some variation). The corresponding regression lines are marked on the graph.[9]

These small growth rates undoubtedly contribute to the strong performance of the algorithm, which we discuss shortly. However, Figure 5 raises another intriguing possibility, which is that the number of nodes that we visit might be *polynomial in the output size*. Indeed, for every triangulation in our test suite, the number of nodes that we visit is at most $10|V|^2$ (where $|V|$ represents the number of solutions). If this were true in general, both the time and space complexity of the tree traversal algorithm would become worst-case polynomial in the combined input and output size. This would be a significant breakthrough for normal surface enumeration. We return to this speculation in Section 7.

We come now to a direct comparison of running times for the tree traversal algorithm and the double description algorithm. Both algorithms are implemented using the topological software package *Regina* [8]. In particular, the double description implementation that we use is already built into *Regina*; this represents the current state of the art for normal surface enumeration, and the details of the algorithm have been finely tuned over many years [12]. Both algorithms are implemented in C++, and all experiments were run on a single core of a 64-bit 2.3 GHz AMD Opteron 2356 processor.

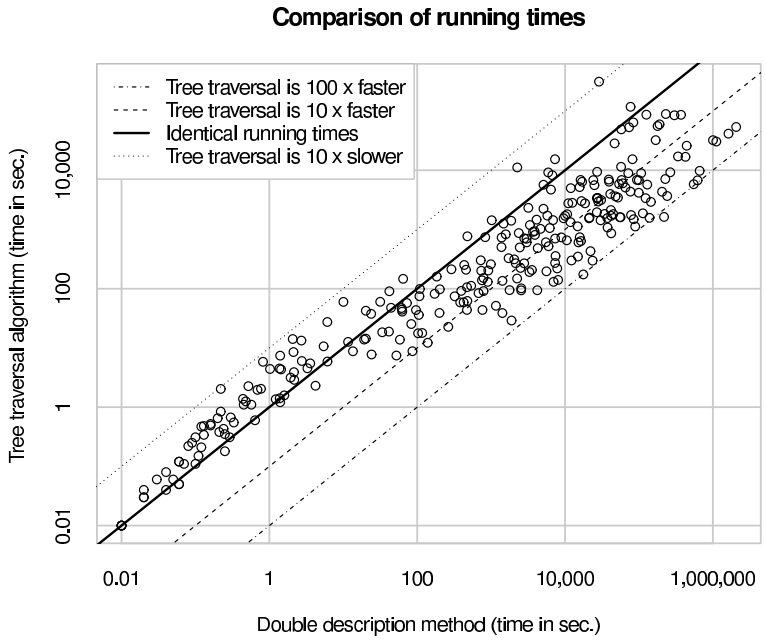

Figure 6: Comparing running times for the old and new algorithms

Figure 6 directly compares the running times for both algorithms: each point on the graph represents a single triangulation from the test suite. Both axes use a logarithmic

[9]These are standard linear regressions of the form $\log y = \alpha n + \beta$, where y is the quantity being measured on the vertical axis.

scale. The solid diagonal line indicates where both algorithms have identical running times, and each dotted line represents an order of magnitude of difference between them. Based on this graph we can make the following observations:

- Problems that are difficult for one algorithm are difficult for both. That is, there are no points in the top-left or bottom-right regions of the graph. This is not always the case for vertex enumeration problems in general [2].
- For smaller problems, the tree traversal algorithm runs slower (in the worst case, around 10 times slower). However, as the problems grow more difficult the tree traversal begins to dominate, and for running times over 100 seconds the tree traversal algorithm is almost always faster (in the best case, around 100 times faster).

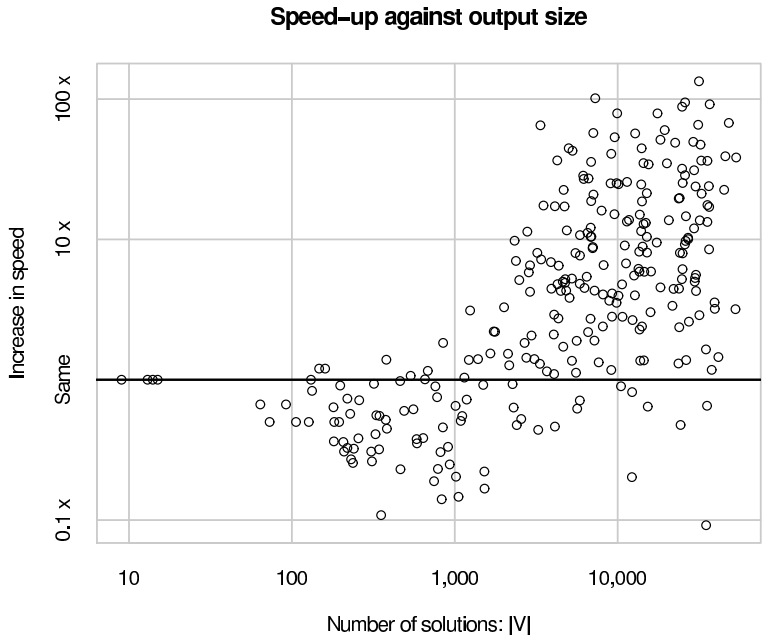

Figure 7: Measuring the performance improvement as the output size grows

Figure 7 plots these speed-up factors explicitly: again each point on the graph represents a single triangulation, and the vertical axis measures the factor of improvement that we gain from the tree traversal algorithm.[10] On the horizontal axis we plot the total number of solutions $|V|$, which is a useful estimate of the difficulty of the problem. Again, both axes use a logarithmic scale.

Here we see the same behaviour as before, but more visibly: for smaller problems the tree traversal algorithm runs slower, and then as the problems grow larger the tree traversal algorithm dominates, running significantly faster in general for the most difficult cases.

7 Discussion

For topological decision problems that require lengthy normal surface enumerations, we conclude that the tree traversal algorithm is the most promising algorithm amongst those

[10]Specifically, the vertical axis measures the double description running time divided by the tree traversal running time.

known to date. Not only does it have small time and space complexities in both theory and practice (though these complexities remain exponential), but it also supports parallelisation, progress tracking, incremental output and early termination. As described in Section 3.2, both parallelisation and incremental output offer great potential for normal surface theory, yet they have not been explored in the literature to date.

One aspect of the tree traversal algorithm that has not been discussed in this paper is the *ordering of tetrahedra*. By reordering the tetrahedra we can alter which nodes of the type tree pass the feasibility test, and thereby reduce the total number of nodes that we visit. Because this total number of nodes is the source of the exponential running time, this simple reordering of tetrahedra could have a great effect on the time complexity (much like reordering the matching equations does in the double description method [2, 12]). Further exploration and experimentation in this area could prove beneficial.

We finish by recalling the empirical observations of Section 6 that the total number of nodes visited—and therefore the overall running time—could be a small polynomial in the combined input and output size. The data presented in Section 6 suggest a quadratic relationship, and similar experimentation on closed triangulations suggests a cubic relationship. Even if we examine triangulations with extremely small output sizes (such as layered lens spaces, where the number of admissible vertices is linear in n [12]), this small polynomial relationship appears to be preserved (for layered lens spaces we visit $O(n^3)$ nodes in total).

Although the tree traversal algorithm cannot solve the vertex enumeration problem for *general* polytopes in polynomial time, there is much contextual information to be gained from normal surface theory and the quadrilateral constraints. Further research into this polynomial time conjecture could prove fruitful: a counterexample would be informative, and a proof would be a significant breakthrough for the complexity of decision algorithms in low-dimensional topology.

Acknowledgements

The authors are grateful to RMIT University for the use of their high-performance computing facilities. The first author is supported by the Australian Research Council under the Discovery Projects funding scheme (project DP1094516).

References

[1] David Avis, *A revised implementation of the reverse search vertex enumeration algorithm*, Polytopes—Combinatorics and Computation (Oberwolfach, 1997), DMV Sem., vol. 29, Birkhäuser, Basel, 2000, pp. 177–198.

[2] David Avis, David Bremner, and Raimund Seidel, *How good are convex hull algorithms?*, Comput. Geom. **7** (1997), no. 5–6, 265–301.

[3] David Avis and Komei Fukuda, *A pivoting algorithm for convex hulls and vertex enumeration of arrangements and polyhedra*, Discrete Comput. Geom. **8** (1992), no. 3, 295–313.

[4] M. L. Balinski, *An algorithm for finding all vertices of convex polyhedral sets*, SIAM J. Appl. Math. **9** (1961), no. 1, 72–88.

[5] Mokhtar S. Bazaraa, John J. Jarvis, and Hanif D. Sherali, *Linear programming and network flows*, 4th ed., Wiley, Hoboken, NJ, 2010.

[6] Joan S. Birman, *Problem list: Nonsufficiently large 3-manifolds*, Notices Amer. Math. Soc. **27** (1980), no. 4, 349.

[7] Robert G. Bland, *New finite pivoting rules for the simplex method*, Math. Oper. Res. **2** (1977), no. 2, 103–107.

[8] Benjamin A. Burton, *Introducing Regina, the 3-manifold topology software*, Experiment. Math. **13** (2004), no. 3, 267–272.

[9] ———, *Converting between quadrilateral and standard solution sets in normal surface theory*, Algebr. Geom. Topol. **9** (2009), no. 4, 2121–2174.

[10] ———, *The complexity of the normal surface solution space*, SCG '10: Proceedings of the Twenty-Sixth Annual Symposium on Computational Geometry, ACM, 2010, pp. 201–209.

[11] ———, *Extreme cases in normal surface enumeration*, In preparation, 2010.

[12] ———, *Optimizing the double description method for normal surface enumeration*, Math. Comp. **79** (2010), no. 269, 453–484.

[13] ———, *Maximal admissible faces and asymptotic bounds for the normal surface solution space*, J. Combin. Theory Ser. A **118** (2011), no. 4, 1410–1435.

[14] Benjamin A. Burton, J. Hyam Rubinstein, and Stephan Tillmann, *The Weber-Seifert dodecahedral space is non-Haken*, Trans. Amer. Math. Soc. **364** (2012), no. 2, 911–932.

[15] David Coulson, Oliver A. Goodman, Craig D. Hodgson, and Walter D. Neumann, *Computing arithmetic invariants of 3-manifolds*, Experiment. Math. **9** (2000), no. 1, 127–152.

[16] Marc Culler and Nathan Dunfield, *FXrays: Extremal ray enumeration software*, `http://www.math.uic.edu/~t3m/`, 2002–2003.

[17] Guy de Ghellinck and Jean-Philippe Vial, *A polynomial Newton method for linear programming*, Algorithmica **1** (1986), no. 4, 425–453.

[18] M. E. Dyer, *The complexity of vertex enumeration methods*, Math. Oper. Res. **8** (1983), no. 3, 381–402.

[19] Komei Fukuda, Thomas M. Liebling, and François Margot, *Analysis of backtrack algorithms for listing all vertices and all faces of a convex polyhedron*, Comput. Geom. **8** (1997), no. 1, 1–12.

[20] Komei Fukuda and Alain Prodon, *Double description method revisited*, Combinatorics and Computer Science (Brest, 1995), Lecture Notes in Comput. Sci., vol. 1120, Springer, Berlin, 1996, pp. 91–111.

[21] Torbjörn Granlund et al., *The GNU multiple precision arithmetic library*, `http://gmplib.org/`, 1991–2010.

[22] Wolfgang Haken, *Theorie der Normalflächen*, Acta Math. **105** (1961), 245–375.

[23] ———, *Über das Homöomorphieproblem der 3-Mannigfaltigkeiten. I*, Math. Z. **80** (1962), 89–120.

[24] Joel Hass, Jeffrey C. Lagarias, and Nicholas Pippenger, *The computational complexity of knot and link problems*, J. Assoc. Comput. Mach. **46** (1999), no. 2, 185–211.

[25] William Jaco, *The homeomorphism problem: Classification of 3-manifolds*, Lecture notes, Available from `http://www.math.okstate.edu/~jaco/pekinglectures.htm`, 2005.

[26] William Jaco, David Letscher, and J. Hyam Rubinstein, *Algorithms for essential surfaces in 3-manifolds*, Topology and Geometry: Commemorating SISTAG, Contemporary Mathematics, no. 314, Amer. Math. Soc., Providence, RI, 2002, pp. 107–124.

[27] William Jaco and Ulrich Oertel, *An algorithm to decide if a 3-manifold is a Haken manifold*, Topology **23** (1984), no. 2, 195–209.

[28] N. Karmarkar, *A new polynomial-time algorithm for linear programming*, Combinatorica **4** (1984), no. 4, 373–395.

[29] Bruce Kleiner and John Lott, *Notes on Perelman's papers*, Geom. Topol. **12** (2008), no. 5, 2587–2855.

[30] Hellmuth Kneser, *Geschlossene Flächen in dreidimensionalen Mannigfaltigkeiten*, Jahresbericht der Deut. Math. Verein. **38** (1929), 248–260.

[31] Sergei Matveev, *Algorithmic topology and classification of 3-manifolds*, Algorithms and Computation in Mathematics, no. 9, Springer, Berlin, 2003.

[32] P. McMullen, *The maximum numbers of faces of a convex polytope*, Mathematika **17** (1970), 179–184.

[33] Nimrod Megiddo, *Improved asymptotic analysis of the average number of steps performed by the self-dual simplex algorithm*, Math. Programming **35** (1986), no. 2, 140–172.

[34] T. S. Motzkin, H. Raiffa, G. L. Thompson, and R. M. Thrall, *The double description method*, Contributions to the Theory of Games, Vol. II (H. W. Kuhn and A. W. Tucker, eds.), Annals of Mathematics Studies, no. 28, Princeton University Press, Princeton, NJ, 1953, pp. 51–73.

[35] J. Hyam Rubinstein, *An algorithm to recognize the* 3*-sphere*, Proceedings of the International Congress of Mathematicians (Zürich, 1994), vol. 1, Birkhäuser, 1995, pp. 601–611.

[36] Steve Smale, *On the average number of steps of the simplex method of linear programming*, Math. Programming **27** (1983), no. 3, 241–262.

[37] Marco Terzer and Jörg Stelling, *Parallel extreme ray and pathway computation*, Parallel Processing and Applied Mathematics, Lecture Notes in Comput. Sci., vol. 6068, Springer, Berlin, 2010, pp. 300–309.

[38] Stephan Tillmann, *Normal surfaces in topologically finite 3-manifolds*, Enseign. Math. (2) **54** (2008), 329–380.

[39] Jeffrey L. Tollefson, *Normal surface Q-theory*, Pacific J. Math. **183** (1998), no. 2, 359–374.

[40] Günter M. Ziegler, *Lectures on polytopes*, Graduate Texts in Mathematics, no. 152, Springer-Verlag, New York, 1995.

Benjamin A. Burton
School of Mathematics and Physics, The University of Queensland
Brisbane QLD 4072, Australia
(bab@maths.uq.edu.au)

Melih Ozlen
School of Mathematical and Geospatial Sciences, RMIT University
GPO Box 2476V, Melbourne VIC 3001, Australia
(melih.ozlen@rmit.edu.au)