Finding the optimal nets for self-folding Kirigami

N. A. M. Araújo,1,2,* R. A. da Costa,3 S. N. Dorogovtsev,3,4 and J. F. F. Mendes3

1Departamento de Física, Faculdade de Ciências, Universidade de Lisboa, 1749-016 Lisboa, Portugal
2Centro de Física Teórica e Computacional, Universidade de Lisboa, 1749-016 Lisboa, Portugal
3Department of Physics & I3N, University of Aveiro, 3810-193 Aveiro, Portugal
4A. F. Ioffe Physics-Technical Institute, 194021 S. Petersburg, Russia

Three-dimensional shells can be synthesized from the spontaneous self-folding of two-dimensional templates of interconnected panels, called nets. The yield is maximized following sequentially two design rules: (i) maximum number of vertices with a single-edge cut and (ii) minimum radius of gyration of the net. Previous methods to identify the optimal net are based on random search and thus limited to very simple shell structures and not guaranteeing a unique solution. Here, we show that the optimal net can be found using a deterministic algorithm. We map the connectivity of the shell into a shell graph, where the nodes and links of the graph represent the vertices and edges of the shell, respectively. Applying the design rule (i) corresponds then to finding the set of maximum leaf spanning trees of the shell graph, to which (ii) can be applied straightforwardly. This method allows not only to designing the self-assembly of much larger shell structures but also to apply additional design rules, as a complete catalog of the maximum leaf spanning trees is obtained.

PACS numbers: 05.65.+b,64.60.aq

The synthesis of three-dimensional polyhedral shells at the micron and nano scales is key for encapsulation and drug delivery [1–3]. Inspired by the Japanese art of Kirigami, where hollowed structures are obtained from cutting and folding a sheet of paper, lithographic methods have been developed to form shells from two-dimensional nanometer templates of interconnected panels [4–8]. The potential is enormous, for a wide range of shapes and sizes can be obtained. Ideally, the unfolded templates (nets) should spontaneously self-fold into the target structure to reduce production costs and achieve large-scale parallel production.

Many nets fold into the same structure, but the time and effectiveness of their self-folding pathways may differ by orders of magnitude. Finding the optimal net that maximizes yield is not simple; The self-folding process depends on the geometry of the net, its physical properties, and its interactions with the surrounding medium [9–11]. Experiments by Pandey et al. suggest that maximum yield is obtained when, from the entire set of nets, one first picks the nets with the maximum number of vertices with a single-edge cut and, from them, the nets with the lowest radius of gyration [12]. A vertex with a single-edge cut is one whose the number of adjacent faces is the same in the net and in the polyhedral shell. The implementation of this global search in Ref. [12] implies considering all possible cuts of the shell. This inefficient procedure is time consuming, technically demanding, and for most shapes impossible, since, as we show here, the number of possible cuts rapidly grows with the number of edges. For example, with only twelve edges, a cube has 384 possible cuts, while a dodecahedron, with thirty edges, has more than 5 million possible cuts. Consequently, previous methods to identify the optimal net of a sufficiently large shell have to be based on random searches in the configuration space. They consider only a subset (actually a small subset) of possible nets and so they do not guarantee a unique, globally optimal solution.

We propose a deterministic procedure to identify the optimal net that only requires generating a subset of all possible cuts. Let us exemplarily consider the case of a cubic shell. As shown in Fig. 1(a), the structure of the shell can be mapped into a shell graph (black nodes and links in the figure), where nodes represent the vertices and links represent the edges. A second graph can also be defined, the face graph, whose nodes are the faces and links connect pairs of adjacent faces (blue graph). Every net of the cubic shell corresponds to a spanning tree of its face graph, i.e., a connected sub-graph that includes all nodes but the minimum number of links (see Fig. 1(c)). The nets can be obtained by a set of cuts along the edges of the shell graph, under the constraint that the set of nodes in the face graph remains connected. The cut is defined as the sub-graph of the shell graph that contains all removed links (cut edges), as represented in red in Figs. 1(b) and (c). It consists of all nodes and it is a spanning tree of the shell graph. The main advantage of the mapping proposed here is that it makes possible to implement in a systematic and deterministic way the two design rules. Below, we discuss how to apply such mapping to 1) identify the cuts that maximize the number of vertices with a single-edge cut and 2) rank them by increasing radius of gyration.

First criterion. The vertices with a single-edge cut are the nodes of unitary degree in the cut, known as the leaves. Since the cut is a spanning tree of the shell graph, the cuts that maximize the number of vertices with a single-edge cut are the maximum leaf spanning trees (MLST) of the shell graph. To identify the full set of MLSTs, we first identify all minimum connected dom-
Figure 4 shows \( L \) as a function of \( E \) for all Platonic and Archimedean solids with up to 150 shell edges. The obtained dependence is consistent with the predicted linear dependence (solid line). From the simple relation (1), one can estimate an upper bound for the number of MLSTs, \( N_{\text{MLST}} \), as a function of \( E \). The exact number of spanning trees, \( N_{\text{ST}} \), is given by the Kirchhoff’s matrix-tree theorem, which states that the total number of labeled spanning trees is given by the product of the eigenvalues \( \lambda_n \) of the Laplacian matrix, i.e.,

\[
N_{\text{ST}} = \frac{1}{V} \prod_{i=1}^{V-1} \lambda_i ,
\]

where \( V \) is the number of vertices and \( \lambda_V = 0 \) is excluded from the product. Nevertheless, we can also get an upper bound for \( N_{\text{ST}} \) as a function of \( E \). Accordingly, as we show in the Supplemental Material [13], an estimate the upper bound for the ratio \( N_{\text{MLST}}/N_{\text{ST}} \) is given by

\[
N_{\text{MLST}}/N_{\text{ST}} \sim 2^{-E/2+3/2} ,
\]

what is also consistent over more than 20 orders of magnitude with the observed exponential dependence shown in Fig. 3. This fast decay with \( E \) reinforces the necessity of a deterministic method, since the chances of obtaining a MLST from a random search is given by this ratio. For example, for the dodecahedron, with only twelve faces, less than 0.04% of its more than 5 million spanning trees are MLSTs, i.e., to obtain a MLST one would need to randomly sample 2500 configurations on average. For the largest shell that we have considered (\( E = 150 \)), this number is larger than \( 10^{26} \). Thus, identifying the optimal net from random methods is practically impossible for such large shells.

With the strategy proposed here, we can also consider open structures, without one or more faces, as the ones shown in Figs. 2(d) and (e). These shells might be relevant for several applications involving, for example, encapsulation, drug delivery, or key-lock mechanism [1–3, 11]. The shell graph for such structures is equal to the corresponding polyhedron; however, every cut includes all edges adjacent to the missing face (connecting two nodes of that face). So, to identify the optimal net, we follow the same procedure as before, but under the constraint that the edges of the missing face of the polyhedron are always in the cut. Note that, in this case, the cut is no longer a tree, as the edges of the missing face form a loop, but this is the only possible loop in the cut as any other will split the net into pieces (see Supplemental Materials for further details [13]).

Second criterion. The second criterion proposed in Ref. [12] is to select, among all possible cuts with the maximum number of vertices with a single-edge cut, the one corresponding to the net with the lowest radius of gyration. To apply this criterion, the label of the individual nodes in the MLST is irrelevant and so we need to identify first the subset of non-isomorphic cuts, i.e., the
FIG. 2. Five examples of shells and of one of their nets corresponding to a cut that is a maximum leaf spanning trees: a) tetrahedron, with four faces and nine edges, it has four maximum leaf spanning trees, but only one non-isomorphic; b) dodecahedron, with twelve faces and thirty edges, it has 1980 maximum leaf spanning trees, but only 21 non-isomorphic; c) small rhombicuboctahedron, with 26 faces and 48 edges, it has 1536 maximum leaf spanning trees, but only 32 non-isomorphic; d) open cubic shell, with five faces and twelve edges, it has only one maximum leaf spanning tree e) small rhombicuboctahedron with the top nine faces removed and 17 faces, 36 edges and 20 nodes remaining, it has 720 maximum leaf spanning trees, but only 90 non-isomorphic. The red circles in the nets indicate the vertices with a single-edge cut.

FIG. 3. Fraction of spanning trees (\(N_{ST}\)) that are maximum leaf spanning trees (\(N_{MLST}\)) as a function of the number of shell edges (\(E\)). This ratio was calculated for a total of 21 shells, including all Platonic solids and all Archimedean solids with up to 150 shell edges (see Table S1 in the Supplemental Material [13]). The solid line corresponds to the estimation given by Eq. (3).

FIG. 4. Number of leaves (\(L\)) as a function of the number of shell edges (\(E\)). This number was calculated for a total of 21 shells, including all Platonic solids and all Archimedean solids with up to 150 shell edges (see Table S1 in the Supplemental Material [13]). The solid line corresponds to the estimation given by Eq. (1).

subset of cuts that one cannot get one from another by relabeling the nodes (or faces). This is a much smaller subset. For example, for the cubic shell, 120 MLSTs were identified, but only four are non-isomorphic. To identify them, we rely on the concept of adjacency matrix \(A_{ij}\) of a graph with \(V\) nodes, defined as \(V \times V\) matrix, where \(A_{ij}\) is either unity, if nodes \(i\) and \(j\) are connected, and zero otherwise. Two graphs are isomorphic if the adjacency matrix of one can be made equal to the other by a set of line and column swaps. Note that, each swap corresponds to a relabeling of the nodes and thus the same swap needs to be done for the lines and the columns.

Figure 5 shows the study for the truncated icosahedron, also known as soccer ball or buckyball. With 60 vertices, 32 faces, and 90 edges, this shell has more than \(10^{20}\) possible cuts. By mapping the shell into a graph, we can identify the 484800 cuts with the maximum num-
number of leaves (30 leaves each) and identify the 4114 non-isomorphic corresponding nets. To calculate the radius of gyration $R_g$ of each net $\alpha \subset \mathbb{R}^2$, we first obtain its centroid, defined as

$$(\bar{x}, \bar{y}) = \frac{1}{A} \int_{\alpha} (x, y) \, dA ,$$

where $A$ is the total area of the shell faces. We then calculate the radius of gyration with respect to the centroid as

$$R_g = \sqrt{\frac{1}{A} \int_{\alpha} [(x - \bar{x})^2 + (y - \bar{y})^2] \, dA} .$$

Figure 5(a) shows the spectrum of the radii of gyration for all the 4114 non-isomorphic nets of the truncated icosahedron, ranked by increasing radius of gyration; Figures 5(b)-(c) show from those nets, the ones with the lowest (optimal), intermediate, and maximum $R_g$. The radius of gyration rapidly increases with the position in the rank. The less optimal net (Fig. 5(d)) has a radius of gyration that is 40% higher than the one of the optimal net (Fig. 5(b)). Previous methods to identify the optimal net are based on random search. Using such methods, even if the obtained net corresponds to one with the maximum number of vertices with a single-edge cut, the probability that its radius of gyration differs by less than 3% from the optimal one is below 1.5% (see inset in Fig. 5(a)). Since the timescale and yield of the self-folding depends strongly on the radius of gyration, the self-folding efficiency of an approximated solution obtained with previous methods based on a random search is likely far from optimal.

**Conclusions.** We proposed a method to identify the optimal net that spontaneously self-folds into a closed or open polyhedral shell structures. The method consists of mapping the shell structure into a shell graph and sequentially apply two design rules: (i) to identify the cuts that maximize the number of vertices with a single-edge cut and, (ii) among those, the one with the minimum radius of gyration. Adapting concepts and methods from Graph Theory, we show that the optimal solution can be obtained in a deterministic and systematic manner. Previous methods are based on random search and thus not providing a unique solution. As we showed, the fraction of cuts that are identified by rule (i) decays exponentially with the number of edges in the polyhedron, reinforcing the necessity of a deterministic method. Also, with the method proposed here, since a complete list of possible cuts respecting rule (i) is obtained, other design rules (alternative to rule (ii)) can be implemented straightforwardly.

It is conjectured that the nets for all convex shells are non-overlapping, a necessary condition for obtaining it from a two-dimensional template. In all examples studied here, all optimal shells are non-overlapping but nets for concave shells are more likely to overlap. In fact, this conjecture is not necessary, for once the optimal net is identified, self-overlap can be tested. If the selected net overlaps, one should proceed through the rank of increasing radius of gyration and pick the first net that does not self-overlap. If all nets for the maximum leaf spanning trees obtained with rule (1) overlap, one should proceed iteratively considering spanning trees with less and less leaves until we found the first that does not overlap.

Identifying the maximum leaf spanning tree is a NP problem and so the numerical complexity will still grow rapidly with the number of shell vertices. If the number of vertices is too large for the straightforward implementation of our deterministic algorithm then, in the spirit of our approach, approximated algorithms can be used that identify spanning trees with a number of leaves that is close to the maximum [17, 18].

We suggest that our deterministic algorithm and its variations could be used when searching for optimal design and production of even more complex self-assembling systems.

We acknowledge financial support from the Portuguese Foundation for Science and Technology (FCT) under Contract no. UID/FIS/00618/2013 and grant
[1] R. Fernandes and D. H. Gracias, “Self-folding polymeric containers for encapsulation and delivery of drugs,” Adv. Drug Deliv. Rev. 64, 1579 – 1589 (2012).

[2] J. Shim, C. Perdigou, E. R. Chen, K. Bertoldi, and P. M. Reis, “Buckling-induced encapsulation of structured elastic shells under pressure,” Proc. Natl. Acad. Sci. 109, 5978–5983 (2012).

[3] M. Filippousi, T. Altantzis, G. Stefanou, M. Betsiou, D. N. Bikiaris, M. Angelakeris, E. Pavlidou, D. Zamboulis, and G. Van Tendeloo, “Polyhedral iron oxide core-shell nanoparticles in a biodegradable polymeric matrix: preparation, characterization and application in magnetic particle hyperthermia and drug delivery,” RSC Adv. 3, 24367–24377 (2013).

[4] D. M. Sussman, Y. Cho, T. Castle, X. Gong, E. Jung, S. Yang, and R. D. Kamien, “Algorithmic lattice kirigami: A route to pluripotent materials,” Proc. Natl. Acad. Sci. 112, 7449–7453 (2015).

[5] Y. Zhang, Z. Yan, K. Nan, D. Xiao, Y. Liu, H. Luan, H. Fu, X. Wang, Q. Yang, J. Wang, W. Ren, H. Si, F. Liu, L. Yang, H. Li, J. Wang, X. Guo, H. Luo, L. Wang, Y. Huang, and J. A. Rogers, “A mechanically driven form of kirigami as a route to 3d mesostructures in micro/nanomembranes,” Proc. Natl. Acad. Sci. 112, 11757–11764 (2015).

[6] A. Lamoureux, K. Lee, M. Shlian, S. R. Forrest, and M. Shtein, “Dynamic kirigami structures for integrated solar tracking,” Nat. Commun. 6, 8092 (2015).

[7] G. P. Collins, “Science and culture: Kirigami and technology cut a fine figure, together,” Proc. Natl. Acad. Sci. 113, 240–241 (2016).

[8] W. M. Jacobs and D. Frenkel, “Self-assembly of structures with addressable complexity,” J. Am. Chem. Soc. 138, 2457–2467 (2016).

[9] G. M. Whitesides and B. Grzybowski, “Self-assembly at all scales,” Science 295, 2418–2421 (2002).

[10] T. G. Leong, P. A. Lester, T. L. Koh, E. K. Call, and D. H. Gracias, “Surface tension-driven self-folding polyhedra,” Langmuir 23, 8747–8751 (2007).

[11] A. Azam, K. E. Laflin, M. Jamal, R. Fernandes, and D. H. Gracias, “Self-folding micropatterned polymeric containers,” Biomed. Microdevices 13, 51–58 (2011).

[12] S. Pandey, M. Ewing, A. Kunas, N. Nguyen, D. H. Gracias, and G. Menon, “Algorithmic design of self-folding polyhedra,” Proc. Natl. Acad. Sci. 108, 19885–19890 (2011).

[13] See Supplemental Material at.

[14] H. Fernau, J. Kneis, D. Kratsch, A. Langer, M. Liedloff, D. Raible, and P. Rossmanith, “An exact algorithm for the maximum leaf spanning tree problem,” Theor. Comput. Sci. 412, 6290–6302 (2011).

[15] T. Fujie, “An exact algorithm for the maximum leaf spanning tree problem,” Comput. Oper. Res. 30, 1931–1944 (2003).

[16] A. Lucena, N. Maculan, and L. Simonetti, “Reformulations and solution algorithms for the maximum leaf spanning tree problem,” Comput. Manag. Sci. 7, 289–311 (2010).

[17] H.-I. Lu and R. Ravi, “Approximating maximum leaf spanning trees in almost linear time,” J. Algorithms 29, 132–141 (1998).

[18] R. Solis-Oba, P. Bonsma, and S. Lowski, “A 2-approximation algorithm for finding a spanning tree with maximum number of leaves,” Algorithmica 77, 374–388 (2017).
S1. ONE-TO-ONE CORRESPONDENCE BETWEEN NETS AND SPANNING TREES

To each net of a polyhedron corresponds a cut along the edges of a spanning tree of the shell graph:

i) To unfold a polyhedral shell into a 2D net, the cut must reach every vertex of the shell graph, and must be connected;

ii) For the polyhedron faces to remain connected as a single component, the cut cannot contain any loops.

The only subgraphs that span to every vertex, are connected, and contain no loops are spanning trees. Therefore, maximizing the number of single-edge cut vertices of a net is equivalent to maximizing the number of leaves of a spanning tree.

S2. THE MAXIMUM LEAF SPANNING TREE

The Maximum Leaf Spanning Tree (MLST) problem has been extensively studied in the scope of graph theory and computer science [S1–S7]. It consist of finding a spanning tree with the largest possible number of leaves in a given undirected unweighted graph. Finding a MLST, or just determining the number of leaves of the MLSTs of a generic graph is a well known NP-complete problem [S1].

Here, we describe a simple (exact) algorithm to find the full set of labeled MLSTs of an arbitrary (undirected and unweighted) graph. Notice that while the algorithms for the MLST problem typically find a single optimal tree [S3–S5], our algorithm provides all possible labeled MLSTs. The number of spanning trees and of MLSTs grow both exponentially with the graph size and the computation time of an algorithm that lists all MLSTs of an arbitrary (undirected and unweighted) graph. Finding the optimal nets for self-folding Kirigami – Supplemental Material

N. A. M. Araújo, R. A. da Costa, S. N. Dorogovtsev, and J. F. F. Mendes

A. Algorithm

In order to list all MLSTs, we use a search algorithm for finding the full set of dominating subtrees with exactly nS vertices (i.e., subtrees with nS vertices that are a connected dominating set of the graph). We start by checking if there are any dominating subtrees of nS = 1 vertices, which only exist when a single vertex is connected to all the other vertices in the original graph. If there is no such tree, we iteratively increase nS by 1 and search again. The search stops when the set of dominating subtrees with nS vertices is non-empty. These minimum dominating subtrees are the interiors of the MLSTs, that is, the MLSTs without the leaf vertices and their respective edges.

To finalize the construction of the MLSTs, we attach the remaining vertices to the obtained minimum dominating subtree. These vertices are the leaves of the MLST. If every leaf vertex only has one edge of the original graph connecting it to the dominating subtree, then there is only one possible MLST with that particular interior subtree. However, some leaf vertices may have multiple edges of the original graph linking them to the dominating subtree, and so there are multiple MLSTs with the same interior subtree. In these cases, we have to chose one of the possibilities for each leaf vertex. Since those choices are independent from each other, the total number of MLSTs that share that particular interior subtree equals the product of the numbers of possibilities of each leaf vertex (i.e., the number of different ways of connecting the leaf vertices to a particular dominating subtree).

The algorithm recursively grows subtrees with nS vertices and nS − 1 edges of the graph starting from a single root vertex and enumerates all MLSTs where the root is a non-leaf vertex. There is a set of vertices, say R, that the algorithm uses as roots (one at each time), and consists of a specific arbitrary vertex and all of its neighbors. For any dominating subtree, all the vertices of the original graph must either be in the subtree or have at least one neighbor in the subtree. To find all MLSTs, it is sufficient to consider the roots in R. In the present work, we use the vertex with the smallest degree and its neighbors as the set of roots R. Given a size of the subtrees nS, the algorithm performs a separate search for each root vertex in R. So, we need an additional constraint to avoid multiple counts of the MLSTs that have more than one non-leaf vertices in R. We introduce a set of vertices, Vexcl, that are explicitly forbidden from joining the dominating subtree. In the first search, rooted in the first vertex of R, say r1, the set
$V_{\text{excl}}$ is empty and the algorithm enumerates all MLSTs where $r_1$ is a non-leaf vertex. Since the vertex $r_1$ must be a leaf in all the other MLSTs still not found, we add $r_1$ to $V_{\text{excl}}$. In the second search, rooted in $r_2$, the vertex $r_1$ is never included in dominating subtree, insuring that the algorithm only returns the MLSTs where $r_1$ is a leaf and $r_2$ is a non-leaf vertex. Then, $r_2$ is added to $V_{\text{excl}}$, the third search is made, and so on.

Given a graph with $V$ vertices labeled $i = 1, \ldots, V$, let us denote the edge that connects the vertices $i$ and $j$ by $e_{ij}$. In the following Algorithms 1-3 the set of vertices that are connected to $i$ in the original graph is denoted $A_i$, and the set $A = \{A_i : i = 1, \ldots, V\}$ provides complete information about the graph.

**Algorithm 1 Enumeration of all MLSTs.** This procedure initializes the necessary variables, and calls the function recursive of Algorithm 2. Each time it is called, recursive enumerates all MLSTs with $L - n_S$ leaves, where the supplied root, $r$, in a non-leaf vertex and the vertices in $V_{\text{excl}}$ are leaves.

```
procedure list_all_mlsts(A)
Input:
List of neighbors, $A_i$, for every vertex $i$ ($A = \{A_i\}$).

$v = \text{arbitrary vertex}$ > For example, a vertex with the
minimum degree
$R = \text{append}(v, A_v)$
$MLSTs = \emptyset$
$n_S = 1$
while $\text{MLSTs} = \emptyset$ do
    $V_{\text{excl}} = \emptyset$
    for all $r \in R$ do
        $E_A = \{e_{ri} : i \in A_r\}$
        $MLSTs' = \text{recursive}(A_r, \text{null}, E_A, V_{\text{excl}}, n_S)$
        $MLSTs = \text{append}(MLSTs, MLSTs')$
        $V_{\text{excl}} = \text{append}(V_{\text{excl}}, r)$
    end for
    $n_S = n_S + 1$
end while
end procedure
```

The procedure list_all_mlsts, shown in Algorithm 1, initializes the set of roots $R$ as described above, and the current size of the searched subtrees, $n_S$, is initialized to 1. The set MLSTs will store the collection of found MLSTs, and starts as an empty set. The search of MLSTs with larger $n_S$, and a smaller number of leaves $L = V - n_S$, will only proceed if the set MLSTs remains empty. The list of vertices $V_{\text{excl}}$ stores the roots of $R$ that were already used for each particular value of $n_S$, and is initialized as an empty set every time that $n_S$ is incremented. The search itself, is performed by the recursive function recursive, Algorithm 2. For each considered $n_S$, the function recursive is called by the procedure list_all_mlsts once for each root in $R$.

The function recursive of Algorithm 2 starts with a single vertex (root) and recursively grows subtrees $T$ up to a predetermined size ($n_S$ vertices), while keeping track of the elements already in the tree and at its border. Let $V_T$ and $E_T$ be the lists of vertices and edges, respectively, currently in $T$, and $E_A$ be the list of edges that are not in $T$ and are connected to at least one vertex in $T$ ($E_A$ is the exterior boarder of $T$, which the algorithm is currently exploring). Furthermore, $V_{\text{excl}}$ is a specific set of vertices that are forbidden to participate in the subtree $T$ (these vertices are the roots of the previous searches for the same $n_S$). When $T$ reaches the target size ($n_S$ vertices) it stops increasing. If the vertices $V_T$ form a dominating set, then the algorithm enumerates the possible ways of joining each vertex outside of $V_T$ to one and only one vertex in $V_T$ by an edge. Each different way of making these last connections represents a different (labeled) spanning tree whose leaves are the vertices outside of $V_T$.

In the first stage, while the number of vertices in the growing subtree $|V_T| < n_S$, the function recursive considers all possibilities for the next edge addition to the subtree $T$ from the set of adjacent edges $E_A$. For each of those possibilities, recursive is called again with the updated $V_T'$, $E_T'$, and $E_A'$. Note that, when an edge in $E_A$ connects two vertices already in $V_T$ it cannot be added to the tree (because it would close a loop in $T$). In order to keep the vertices in the set $V_{\text{excl}}$ outside of $T$, the edges that lead to those vertices are not added to $E_A$ in the update. In this way, recursive finds all configurations of subtrees with $n_S$ vertices that include the root and exclude all the vertices in $V_{\text{excl}}$. In the second stage, when finally $|V_T| = n_S$, recursive checks if the vertices in $V_T$ form a dominating set: if so, it finishes the construction of the spanning trees by connecting the leaves in every possible way, otherwise it returns an empty set.

In the search algorithm presented here, we consider only connected sets of vertices (subtrees), and check if they are dominating. Furthermore, we only consider subtrees that include either a specific arbitrary vertex or at least one of its neighbors (if a subtree is dominating, every vertex of the graph fulfills this requirement). The combination of these two strategies drastically reduces the configurational space and computation time.
Algorithm 2 Recursive search function. This function generates all possible subtrees \( T \) with \( n_S \) vertices that include the root vertex and exclude all the vertices in \( V_{excl} \). If \( T \) is a dominating subtree, then the function lists all possible spanning trees that can be obtained by joining the remaining vertices of the graph to \( T \).

```
function recursive(A, V_T, E_T, E_A, V_{excl}, n_S)
Input:
List of neighbors, \( A_i \), for every vertex \( i \) (\( A = \{ A_i \} \)).
Lists of vertices, \( V_T \), and edges, \( E_T \), currently in \( T \).
List of edges, \( E_A \), currently adjacent to \( T \).
List of vertices, \( V_{excl} \), excluded from \( T \).
Final number of vertices, \( n_S \), in \( T \).
Output:
List of spanning trees, \( STs \), with at least \( V - n_S \) leaves (each spanning tree is returned as the list of its edges).

\[
STs = \emptyset
\]

if \( |V_T| < n_S \) then \( \triangleright \) Add one edge to \( T \).

for all \( e_{jk} \in E_A \) do
  \( E_A = E_A \setminus \{ e_{jk} \} \)
  if \( k \notin V_T \) then
    \( i = k \)
  else if \( j \notin V_T \) then
    \( i = j \)
  else
    continue
  end if
  \( E_T' = append(E_T, e_{jk}) \)
  \( V_T' = append(V_T, i) \)
  \( E_A' = E_A \)
  for all \( l \in A_i \) do
    if \( e_{il} \notin E_T \) and \( l \notin V_{excl} \) then
      \( E_A' = append(E_A', e_{il}) \)
    end if
  end for
  \( STs' = recursive(A, V_T', E_T', E_A', V_{excl}, n_S) \)
  \( STs = append(STs, STs') \)
end for
\[
return STs
\]
else if \( V_T \) is a dominating set then \( \triangleright \) Expand \( T \) into corresponding spanning trees.

\( STs = \{ E_T \} \)

for all vertices \( i \notin V_T \) do
  \( STs' = \emptyset \)
  for all vertices \( j \in A_i \cap V_T \) do
    for all trees \( U \in STs \) do
      \( STs' = append(STs', append(U, e_{ij})) \)
    end for
  end for
  \( STs = STs' \)
end for
\[
return STs
\]
else
  \( return \emptyset \)
end if
end function
```

The order by which the edges of \( E_A \) are picked in the outermost for-cycle of the search of recursive is not specified in Algorithm 2 because any order will produce the same result. In our implementation of the function recursive, we picked the edges in \( E_A \) using a first-in-first-out method and effectively performed a breadth-first search. If, for instance, the edges of \( E_A \) are picked in a last-in-first-out fashion, then the algorithm will perform depth-first search instead.

The algorithm described in this section for listing all MLSTs is defined for labeled graphs, and finds the set of labeled MLSTs. If the original graph has automorphisms, i.e., a relabeling that results in the same labeled graph, then the set of labeled MLSTs may have multiple ‘copies’ of the same unlabeled MLST with different labelings (isomorphic MLSTs). In this work, we consider the nets of polyhedra with high regularity, such as Platonic and Archimedean solids where all vertices are equivalent. The polyhedral graphs of polyhedra where all vertices are equivalent, or with sets of equivalent vertices, have automorphisms. However, each geometrically distinct net is entirely determined by a cut of the polyhedral graph along the edges of an unlabeled spanning tree. So, to determine the number of distinct nets, we need to disregard the labels of MLSTs, and search for isomorphisms in the set of MLSTs found by the algorithm. To check if any two labeled MLSTs are isomorphic, we simply check if there is an automorphic relabeling of the original graph that maps one labeled MLST into the other, see Section for details.

Typically the algorithms for the MLST problem are designed to find the number of leaves in a MLST, and not the full set of MLSTs [S2–S5]. Some of these algorithms include multiple stages of optimization, which include heuristics, the use of approximated algorithms to make initial guesses, and other sophisticated approaches. Due to the complexity of this problem, and to the best of our knowledge, the fastest existing algorithms can find one MLST only in graphs with up to roughly 200 vertices [S8].

In this work, we find not just one MLST, but we list also the full set of optimal nets (unlabeled MLSTs) for polyhedral graphs with up to 60 vertices. The number of different optimal nets strongly depends on the details of the polyhedral graph. For instance, the Truncated Icosahedron and the Truncated Dodecahedron each have 60 vertices, 32 edges, and 90 faces, however their numbers of optimal nets are 4 114 and 3 719 677 167, respectively. Table S1 summarizes the exact results obtained in this work with the algorithm presented here, namely the number of leaves in a MLST and the number of optimal nets of each of the 21 polyhedron considered. Table S1 also shows a figure of the optimal net which minimizes the radius of gyration for each polyhedra.

Finally, it should be mentioned that there are several approximated algorithms that find spanning trees with a high number of leaves, close to the maximum possible. If a polyhedral graph is too large to solve by exact methods, these approximated algorithms can find a near-optimal
solution in linear or almost linear time [S6, S7].

S3. SHELLS WITH HOLES

We consider now the problem of finding optimal nets for shells that contain holes, i.e., shells consisting of all the faces of a polyhedron except for one. The graph of the vertices and edges of the shell is the same as the polyhedral graph of the complete polyhedron. The difference is that all the edges adjacent to the missing face will be in every cut, effectively detaching that face from the rest of the net, as intended. (An edge adjacent to a face is an edge between two vertices of that face.) For this reason, a vertex adjacent to the missing face cannot be a single-edge cut vertex in the net, since it always has two edges included in the cut.

The subgraph of the cut edges in the presence of a hole is not a pure tree, because it contains a single loop formed by the edges adjacent to the hole. For the shell to unfold into a 2D net, the cut subgraph must reach all vertices (spanning) and be connected. Also, for the shell’s faces to remain connected in a single component, the cut subgraph cannot have any other loop apart from one surrounding the hole. Then, the cut subgraph consists of the loop adjacent to the hole, and some loopless branches connected to it.

We use the following Algorithm 3 to maximize the number of single-edge cut vertices in nets of shells that contain holes. Algorithm 3 is a simple adaptation of Algorithm 1 that calls the same recursive function of Algorithm 2, gradually increasing the allowed number of non-leaf vertices \( n_S \). The recursive function remains unchanged in this procedure. In this version of the algorithm, in addition to the lists of adjacencies, \( A = \{A_i\} \), we supply the list of vertices adjacent to the hole, \( V_h \). Instead of single root vertex, the search starts from a subgraph already containing all the vertices and edges adjacent to the hole (these vertices and edges must be present in all the cuts, optimal or not). Then, in the first call of the function recursive for each \( n_S \), \( V_T = V_h \) and \( E_T \) is the set edges adjacent to the hole. The set \( E_A \) is initialized with the edges connected to the vertices of \( V_h \) that are not in \( E_T \), as in Algorithm 1. For the sake of clarity, while in Algorithm 1 we denote the set of edges of the optimal cuts by \( MLSTs \), in Algorithm 3 we denote it by \( Cuts \) because in the presence of holes the cuts contain a loop surrounding the hole, and they are no longer trees. When Algorithm 3 calls the function recursive, it passes an empty set as the fifth input argument. This set is the list of vertices that are forced to be leaves, called \( V_{excl} \) in recursive, which we do not use in this algorithm.

| Polyhedron               | \( V \) | \( F \) | \( E \) | \( L \) | \( N_{opt nets} \) | Min RG |
|--------------------------|--------|--------|--------|--------|-------------------|--------|
| Tetrahedron              | 4      | 6      | 4      | 3      | 1                 |        |
| Octahedron               | 6      | 8      | 12     | 4      | 2                 |        |
| Cube                     | 8      | 6      | 12     | 4      | 4                 |        |
| Icosahedron              | 12     | 20     | 30     | 8      | 21                |        |
| Dodecahedron             | 20     | 12     | 30     | 10     | 21                |        |
| Octagonal Piramid        | 9      | 9      | 16     | 8      | 1                 |        |
| Octagonal Dipiramid      | 10     | 16     | 24     | 7      | 3                 |        |
| Truncated Tetrahedron    | 12     | 8      | 18     | 6      | 4                 |        |
| Cuboctahedron            | 12     | 14     | 24     | 7      | 34                |        |
| Truncated Cube           | 24     | 14     | 36     | 10     | 399               |        |
| Snub Cube                | 24     | 38     | 60     | 16     | 600               |        |
| Rhombicuboctahedron      | 24     | 26     | 48     | 15     | 32                |        |
| Truncated Octahedron     | 24     | 14     | 36     | 12     | 56                |        |
| Icosidodecahedron        | 30     | 32     | 60     | 16     | 308 \( 928 \)     |        |
| Truncated Cuboctahedron  | 48     | 26     | 72     | 24     | 244               |        |
| Truncated Icosahedron    | 60     | 32     | 90     | 30     | 4 114             |        |
| (soccer ball)            |        |        |        |        |                   |        |
| Truncated Dodecahedron   | 60     | 32     | 90     | 22     | 3 719 \( 677 167 \) |        |
| Rhombicosidodecahedron   | 60     | 62     | 120    | 37     | 77 \( 952 \)      |        |
| Snub Dodecahedron        | 60     | 92     | 150    | 39     | 13 436 \( 928 \)  |        |
| Triakis Icosahedron      | 32     | 60     | 90     | 26     | 664 \( 128 \)     |        |
| Pentakis Dodecahedron    | 32     | 60     | 90     | 22     | 845 \( 280 \)     |        |

TABLE S1. Optimal nets. The number of leaves, \( L \), in the MLSTs, and the number of distinct optimal nets (unlabeled MLSTs), \( N_{opt nets} \), were obtained for each polyhedron with our algorithm. The numbers of vertices \( V \), faces \( F \), and edges \( E \) are also shown, as well as the optimal net with the smallest radius of gyration (only for cases with \( N_{opt nets} < 10000 \)). The red circles indicate the single-edge cut vertices.
Algorithm 3: Enumeration of all optimal cuts of shells with a hole. This procedure initializes the necessary variables and calls the recursive function of Algorithm 2. The set Cuts stores the spanning subgraphs that include all edges adjacent to the hole and maximize the number of leaves.

procedure list_cuts_with_hole(A, V_h)  
Input:
Lists of neighbors, A_i, of each vertex i (A = {A_i}). Vertices of the hole V_h.

Cuts = ∅  
n_S = |V_h|
while Cuts = ∅ do  
  E_T = e_{i,j} : i, j ∈ V_h  
  E_A = e_{i,j} : j ∈ A_i and i ∈ V_h and j /∈ V_h  
  Cuts’ = recursive(A, V_h, E_T, E_A, ∅, n_S)  
  Cuts = append(Cuts, Cuts’)  
n_S = n_S + 1
end while  
end procedure

S4. ESTIMATIONS

A. Maximum number of leaves

The aim of this section is to estimate the number of leaves L in a MLST in terms of the simplest possible polyhedral parameters, preferably as a function of the number of vertices V, or of edges E. The exact number L depends on the details of the graph, and its determination requires to actually solve the maximum leaf spanning tree problem. However, we can obtain a simple estimate for L by considering a local optimization algorithm for finding approximated maximum leaf spanning trees.

This algorithm iteratively grows a tree by progressively attaching to it vertices of the graph, until the tree becomes spanning (i.e. reaches all vertices in the polyhedral graph), in the following way:

(i) To seed the iterative process, connect the highest degree vertex to all its neighbors.

(ii) If the current tree is a spanning tree the algorithm reaches its end. Otherwise, from the vertices already in the tree, select the one with the highest number of neighbors still not in the tree, connect it to those neighbors, and repeat step (ii).

In this algorithm, except for the highest degree vertex of step (i), every vertex added to the tree starts as a leaf attached to a non-leaf vertex. The neighbors of non-leaf vertices of the intermediate (non-spanning) trees are guaranteed to be in the tree as well. Furthermore, the number of leaves in the final spanning tree is \( L = V - n_{\text{iter}} - 1 \), where \( n_{\text{iter}} \) is the total number of iterations, and the -1 is due to the initial step (i). The number \( n_{\text{iter}} \) depends on how many vertices are added to the tree in each iteration.

As we show in the following, the number of vertices added to the tree at each iteration turns out to be close to 2, regardless of the details of the polyhedral (shell) graph. If the polyhedron has no triangular faces, a vertex of degree \( k \) selected in step (ii) contributes at most with \( k-1 \) new leaves, because each vertex in the tree has at least one neighbor also in the tree. However, if the faces are triangular, any two vertices connected to each other share two common neighbors. In this case, the selected vertex has at least 3 neighbors already in the tree, namely, the parent non-leaf vertex plus two vertices that are common neighbors with the parent vertex, therefore it contributes at most with \( k - 3 \) new leaves.

In convex polyhedra with regular faces (equilateral triangles, squares, regular pentagons, etc), the sum of the internal angles attached to a vertex must be smaller than \( 2\pi \). This strongly constrains the types of faces that can be attached to a vertex of degree \( k \), and in particular the number of triangles. Notice that \( k \) cannot be smaller than 3, and that \( k = 6 \) is not feasible even with just triangles (in that case the sum of angles is equal to \( 2\pi \)). Then, let us consider the number of new vertices added to the tree when the vertex selected in step (ii) has degree \( k = 3, 4, \) and 5. A vertex with \( k = 5 \) is mainly surrounded by triangles (at least 4 out of 5 faces must be triangles), so, when selected in step (ii), most of the times it can contribute with \( k - 3 = 2 \) new leaves to the tree. A vertex with \( k = 4 \) must have between 1 and 4 triangles among its faces, and it can contribute with 1 to 3 new leaves depending on the particular configuration of the faces. Finally, a vertex with \( k = 3 \) can have any number of triangles between 0 and 3. On the one hand, if it has 3 triangles, this vertex is never selected in step (ii) because it has 0 = \( k - 3 \) neighbors still not in the tree. On the other hand, if the vertex has no triangular faces attached, it can contribute with \( k - 1 = 2 \) new leaves.

Due to these geometrical constraints, the average number of new vertices added to the tree at each iteration is essentially independent of the size and local details of the polyhedral graph, and is close to 2. Furthermore, we expect these arguments to qualitatively hold also for irregular convex polyhedra. Even when the internal angles of a face are different from each other, their sum is the same, and so, on average, the internal angles are the same as for a regular face.

Assuming that each iteration adds approximately 2 new vertices on average to the growing tree, and that the tree becomes spanning after \( n_{\text{iter}} \) iterations, we can write:

\[
k_0 + 1 + 2n_{\text{iter}} \sim V, \tag{S1}
\]

where \( k_0 \) is the highest degree in the polyhedral graph. For simplicity, let us use \( k_0 = 4 \). Replacing \( n_{\text{iter}} = V -
\(L - 1\) in Eq. (S1) we get
\[ L \sim (V + 3)/2. \tag{S2} \]
This formula fits well with the general trend observed for \(L\) vs. \(V\), as shown in the inset of Fig. S1.

Interestingly, the main panel of Fig. S1 shows that the dispersion of the points is significantly smaller if we plot \(L\) vs. \(E\). Let us recall Euler’s polyhedron formula \(V + F = E + 2\), where \(V\), \(F\), and \(E\) are the numbers of vertices, faces, and edges of the polyhedron, respectively. For a given \(E\) there are different polyhedra with different combinations of \(V\) and \(F\) such that \(V + F = E + 2\). On the one hand, if all faces are triangles we have \(F = 2E/3\) and in this case \(V = E/3 + 2\). On the other hand, if most faces of the polyhedron have many edges (which implies wide internal angles) most vertices are attached to only three faces, since the sum of the angles must be smaller than \(2\pi\) for convex polyhedra. Then, in this case \(V \approx 2E/3\). Taking this into account, and because we consider a variety of polyhedra with different types of faces, we use the middle point,
\[ V \sim E/2 + 1, \tag{S3} \]
as an estimation of the number of vertices \(V\) of a polyhedron with \(E\) edges. We replace \(E/2 + 1\) for \(V\) in Eq. (S2), and finally obtain
\[ L \sim E/4 + 2. \tag{S4} \]
This formula, plotted in the main panel of Fig. S1 as a solid line, shows a remarkable agreement with our results.

### B. Ratio \(N_{\text{MLST}}/N_{\text{ST}}\)

We now estimate the ratio \(N_{\text{MLST}}/N_{\text{ST}}\), between the the numbers of maximum leaf spanning trees, \(N_{\text{MLST}}\), and of spanning trees, \(N_{\text{ST}}\), in terms of the number of edges, \(E\), of a labeled polyhedral graph. These numbers grow very quickly with the size of the graph and are not uniquely determined by \(E\) – they depend on the details of the graph. To obtain a simple estimate, we calculate upper bounds for \(N_{\text{ST}}\) and \(N_{\text{MLST}}\), and use their ratio as an estimator.

Each spanning tree has \(V - 1\) edges out of a total of \(E\) edges. Therefore, the upper bound of the number of spanning trees \(N_{\text{ST}}\) is the number of possible combinations of \(V - 1\) edges, given by the binomial coefficient \(\binom{E}{V - 1}\). Similarly, each maximum leaf spanning tree has \(L\) leaves of a total of \(V\) vertices, and the upper bound for \(N_{\text{MLST}}\) is the number of combinations of \(L\) vertices, \(\binom{V}{L}\).

To obtain an estimation for \(N_{\text{MLST}}/N_{\text{ST}}\), we take the ratio of the upper bounds and replace the approximations for \(L\) and \(V\) of Eqs. (S2) and (S3), respectively:
\[ N_{\text{MLST}}/N_{\text{ST}} \sim \frac{(E/2 + 1)}{(E/4 + 2)} \sim 2^{-E/2 + 3/2} \tag{S5} \]
where we used Stirling’s approximation \(n! \sim \sqrt{2\pi n}(n/e)^n\). Figure S2 clearly shows that the ratio \(N_{\text{MLST}}/N_{\text{ST}}\) has an exponential-like decay with the number of edges of the polyhedron. The simple form of Eq. (S5) fits well with this decay.

### S5. NON-ISOMORPHIC CUTS

To determine the optimal nets of a polyhedron, which have no labels, we need to find the set of distinct unlabeled MLSTs. However, the algorithm of Section distinguishes each node by its label, and does not consider symmetries that may exist in the polyhedral graph (automorphisms). An automorphism of a labeled graph is
a relabeling that results in the same graph. If the polyhedral graph contains automorphisms, then the set of MLSTs may contain isomorphisms, i.e., multiple ‘copies’ (differently labeled) of the same unlabeled MLST [S8]. These isomorphic cuts correspond to nets that are indistinguishable from each other and have the same radius of gyration, which is our second criterion of optimization. Therefore, we need only one member of each set of isomorphic MLSTs in the list of optimal cuts. To determine if two MLSTs are isomorphic we employ an adjacency matrix approach. (Note that this approach is valid for any isomorphic subgraphs, not just MLSTs.)

The adjacency matrix, $A$, is a convenient representation of labeled graphs. Each element $A_{ij}$ is 1 if the vertices with labels $i$ and $j$ are connected by an edge, otherwise $A_{ij}$ is 0. To switch the labels of any pair of vertices, say $i$ and $j$, we simply switch the rows $i$ and $j$ and the columns $i$ and $j$ in matrix $A$. Any permutation of $V$ labels can be mapped into any other by a series of at most $V-1$ switches between pairs of vertices ($V$ is the number of vertices). We find the complete set of automorphisms of the polyhedral graph by comparing the relabeled adjacency matrix $A'$ with the original matrix $A$: when $A' = A$ the relabeling is automorphism.

Similarly to the polyhedral graph, we can represent a MLST by an adjacency matrix $B$ where each element $B_{ij}$ is 1 if vertices $i$ and $j$ are connected by an edge in the MLST and 0 otherwise. Two MLSTs, or cuts, are isomorphic if, and only if, there is an automorphism of the polyhedral graph that maps one MLST into the other. That is, if two cuts, with adjacency matrices $B_1$ and $B_2$, unfold into the same net, then there are automorphic relabelings of $A$ that map $B_1$ into $B_2$ and vice-versa. We apply each of the previously obtained automorphisms of $A$ to one of the matrices, say $B_2$, and compare the relabeled matrix $B_2'$ with $B_1$. If one of those relabeling gives $B_2' = B_1$, then the two cuts are isomorphic, and we may discard one of them. We systematically compare each of the MLSTs remaining in the list with all the others to eliminate any isomorphisms, and obtain the full set of $N_{opt} \text{net}$ distinct optimal nets of the polyhedron.

**S6. NUMBER OF LABELED SPANNING TREES**

Kirchhoff’s matrix-tree theorem allows us to calculate the exact number of spanning trees of a labeled graph, $N_{ST}$, in terms of the spectrum of the Laplacian matrix. The Laplacian matrix of a graph is defined as $L = D - A$, where $D$ is the degree matrix (i.e., the diagonal matrix with each entry $d_{ii}$ equal to the degree of vertex $i$), and $A$ is the adjacency matrix. For a graph of $V$ vertices the matrix $L$ has $V$ eigenvalues $\lambda_i \geq 0$, the smallest of which is $\lambda_V = 0$. The matrix-tree theorem states that the total number of spanning trees, $N_{ST}$, is given by the product of eigenvalues of its Laplacian matrix

$$N_{ST} = \frac{1}{V} \prod_{i=1}^{V-1} \lambda_i,$$

where $\lambda_V$ is excluded from the product, and $\lambda_i > 0$ for connected graphs and $i < V$ [S8, S9]. The values of $N_{ST}$ used to plot the points in Fig. S2 (Fig. 3 of the main text) were obtained with Eq. (S6) (while the values of $N_{MLST}$ were obtained by the algorithm described in Section ).

| $V$  | $E$  | $N_{nets}$ | $N_{ST}/N_{aut}$ | $\Delta$ |
|------|------|------------|------------------|---------|
| Tetrahedron | 4 | 6 | 2 | 2/3 | 0.66 |
| Cube | 8 | 12 | 11 | 8 | 0.27 |
| Octahedron | 6 | 12 | 11 | 8 | 0.27 |
| Icosahedron | 12 | 30 | 43,380 | 43,200 | 0.0041 |
| Dodecahedron | 20 | 30 | 43,380 | 43,200 | 0.0041 |
| 5-cell | 5 | 10 | 3 | 25/24 | 0.65 |
| 8-cell | 16 | 32 | 261 | 216 | 0.17 |
| 16-cell | 8 | 24 | 110,912 | 110,502 | 0.0029 |
| 24-cell | 24 | 96 | $1.79 \times 10^{16}$ | $1.79 \times 10^{16}$ | $1.9 \times 10^{-11}$ |
| 120-cell | 600 | 1200 | $2.76 \times 10^{119}$ | $2.76 \times 10^{119}$ | $3.9 \times 10^{-61}$ |
| 600-cell | 120 | 720 | $1.20 \times 10^{407}$ | $1.20 \times 10^{407}$ | 0 |

TABLE S2. Comparison of the exact number of nets $N_{nets}$ [S8] and the estimation $N_{ST}/N_{aut}$ for all regular convex polytopes in 3D (top 5 rows) and 4D (bottom 6 rows). The right-hand side column of the table is the relative difference $\Delta = (N_{nets} - N_{ST}/N_{aut})/N_{nets}$. The numbers of vertices, and edges are denoted $V$ and $E$, respectively. We include the 4D polytopes in this table to show that $\Delta$ approaches 0 very quickly as the size ($V$ and $E$) of the polytope increases. Note that everywhere else in this paper we consider only 3D shells and their 2D nets.

*The number of nets of the 600-cell shown in Ref. [S8] is wrong, due to a mistake in the calculation of the graph’s spectrum. Correcting the mistake gives exactly $2^{302}.5^{102}.7^{36}.11^{48}.23^{38}.29^{36}$ nets for the 600-cell. In this particular case, the lower-bound and the actual number of nets coincide exactly.

**S7. NUMBER OF NON-ISOMORPHIC CUTS**

The number of non-isomorphic cuts (i.e. nets), $N_{nets}$, of a polyhedron with no automorphisms is equal to $N_{ST}$. When the polyhedral graph has automorphisms, the number of distinct nets is actually smaller than $N_{ST}$, see Section . In that case, the exact number of nets, $N_{nets}$, can be obtained using the approach of Ref. [S8], which involves a detailed, case by case, analysis for each polyhedron. In Ref. [S8] the $N_{nets}$ are obtained only for the five platonic solids. Nevertheless, we can estimate $N_{nets}$ for other polyhedron with automorphisms (e.g., Archimedean solids) with high precision by taking the ratio of $N_{ST}$ to the number of automorphisms of the graph, $N_{aut}$. The ratio $N_{ST}/N_{aut}$ is in fact a lower-bound of $N_{nets}$: This ratio assumes that each unlabeled spanning tree contributes with $N_{aut}$ differently labeled ‘copies’ to
the set of labeled spanning trees, however, some spanning trees have a smaller number of isomorphic ‘copies’, due to the existence of symmetries in their branches. Since $N_{\text{aut}}$ can be found with linear time algorithms [S10], the calculation of $N_{ST}/N_{\text{aut}}$ is straightforward.

Table S2 clearly shows that this lower-bound actually approaches $N_{\text{nets}}$ very quickly for large graphs. This happens because the fraction of spanning trees that have some symmetry in their structure quickly approaches 0 as the size of the graph increases. Figure S3(a) demonstrates that the number of distinct nets, $N_{nets} \approx N_{ST}/N_{\text{aut}}$, grows exponentially with the size of the polyhedral graph.

![Graph 3(a)](image)

FIG. S3. Precise estimations of (a) the number of distinct nets $N_{nets} \approx N_{ST}/N_{\text{aut}}$, and of (b) the fraction of optimal nets $N_{opt \ nets}/N_{nets} \approx N_{opt \ nets}N_{\text{aut}}/N_{ST}$ vs. the number of edges $E$ of the 3D polyhedra in Table S1. The number $N_{nets}$ grows exponentially with $E$, and shows a remarkably low level of dispersion. The solid line in the plot of panel (b) is Eq. (S5). The fraction of unlabeled optimal cuts $N_{opt \ nets}/N_{nets}$ is essentially equal to the fraction of labeled optimal cuts, $N_{MLST}/N_{ST}$, see Fig. S2.

The probability that a randomly sampled labeled spanning tree is a MLST is equal to the ratio $N_{MLST}/N_{ST}$, which is shown is Fig. 3 of the main text, and reproduced in this Supplemental Material as Fig. S2. If we were to generate nets with a method that avoids the sampling of isomorphic cuts completely, thus reducing the search space, the probability that a randomly sampled net has the maximum number of single-edge cut vertices, $N_{opt \ nets}/N_{nets}$, would still be essentially the same as $N_{MLST}/N_{ST}$.

Figure S3(b) shows the probability that a net sampled at random from the set of non-isomorphic nets has the maximum number of single-edge cut vertices, i.e., $N_{opt \ nets}/N_{nets} \approx N_{opt \ nets}N_{\text{aut}}/N_{ST}$. We observe no significant differences between Figs. S2 and S3(b), which means that searches over unlabeled and labeled configurations have similar performances when only a small fraction of the configurations is sampled.

* nmarauo@fc.ul.pt

[S1] M. R. Garey and D. S. Johnson, *Computers and intractability*, Vol. 29 (W. H Freeman and Company, New York, 2002).

[S2] F. Rosamond, “Max leaf spanning tree,” in *Encyclopedia of Algorithms* (Springer, 2008) pp. 1–99.

[S3] H. Fernau, J. Kneis, D. Kratsch, A. Langer, M. Liedloff, D. Raible, and F. Rossmannith, “An exact algorithm for the maximum leaf spanning tree problem,” Theor. Comput. Sci. 412, 6290–6302 (2011).

[S4] T. Fujie, “An exact algorithm for the maximum leaf spanning tree problem,” Comput. Oper. Res. 30, 1931–1944 (2003).

[S5] A. Lucena, N. Maculan, and L. Simonetti, “Reformulations and solution algorithms for the maximum leaf spanning tree problem,” Comput. Oper. Res. 37, 289–311 (2010).

[S6] R. Solis-Oba, P. Bonsma, and S. Lowski, “A 2-approximation algorithm for finding a spanning tree with maximum number of leaves,” Algorithmica 77, 374–388 (2017).

[S7] H.-I. Lu and R. Ravi, “Approximating maximum leaf spanning trees in almost linear time,” J. Algorithms 29, 132–141 (1998).

[S8] F. Buekenhout and M. Parker, “The number of nets of the regular convex polytopes in dimension $\leq 4$,” Discrete Math. 186, 69–94 (1998).

[S9] D. M. Cvetković, M. Doob, and H. Sachs, *Spectra of Graphs: Theory and Applications, 3rd Ed.* (Wiley, New York, 1998).

[S10] J. E. Hopcroft and J.-K. Wong, “Linear time algorithm for isomorphism of planar graphs (preliminary report),” in *Proc. 6th Annual ACM Symp. Theory of Computing* (ACM, 1974) pp. 172–184.