SUPPLEMENTARY INFORMATION

Increasing topological diversity during computational "synthesis" of porous crystals: how and why

Ryther Anderson[a] and Diego A. Gómez-Gualdrón[a] *

[a] Department of Chemical and Biological Engineering, Colorado School of Mines, Goden CO 80401, USA

* dgomezgualdron@mines.edu

Table of contents

Section S1. Details on ToBaCCo implementation

■ Graphs and embeddings

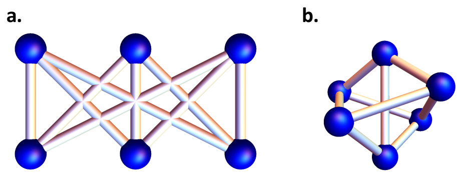

Figure S1. Example of faithful vs. unfaithful embedding of a graph. a) The $K_{3,3}$ graph (known as the utility graph) unfaithfully embedded in $\mathbb{R}^2$. Notice that edges cross as they lie in the same plane, b) The $K_{3,3}$ graph faithfully embedded in $\mathbb{R}^3$. Notice that edges do not cross.

■ On why topological blueprints need to be rescaled anisotropically

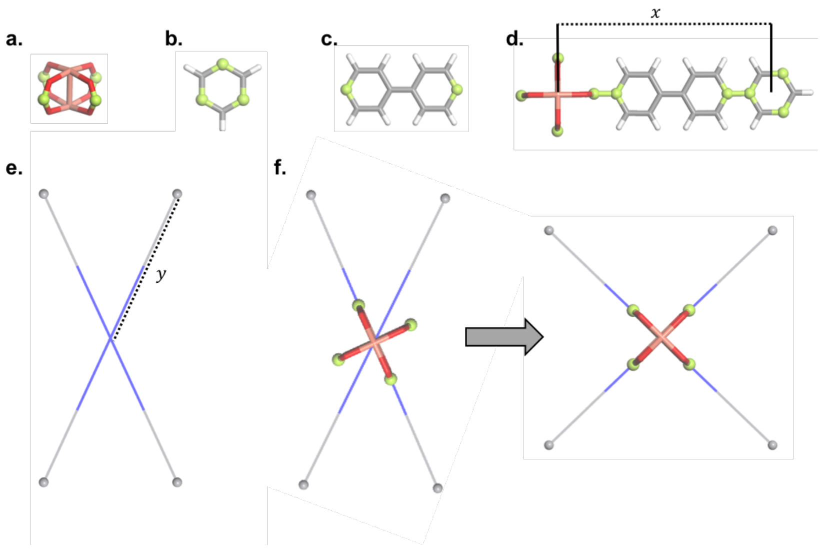

Figure S2. Building blocks and illustration of the need for anisotropic rescaling to properly map a building block onto a node in a topological blueprint. Connection sites are highlighted in yellow for all shown building blocks. a) 4-connected Cu-Paddlewheel (a nodular building block), b) 3-connected phenyl (a nodular building block), C) a 2-connected diphenyl (a connecting building block), and d) how building blocks connect with each other during MOF computational synthesis (the anticipated value of "x" is used in rescaling procedures). e) The vertex geometry of 4-connected nodes in the barycentric representation of the **sty** topology. The vertex itself is represented in blue and the connected vertices are presented as gray points (the value of "y" is used in rescaling procedure). f) how anisotropic rescaling of the net allows the node geometry to better fit the Cu-paddlewheel, which in turn allows the paddlewheel to be properly mapped onto the node. (note that the rescaling is such that "y" becomes equal to "x").

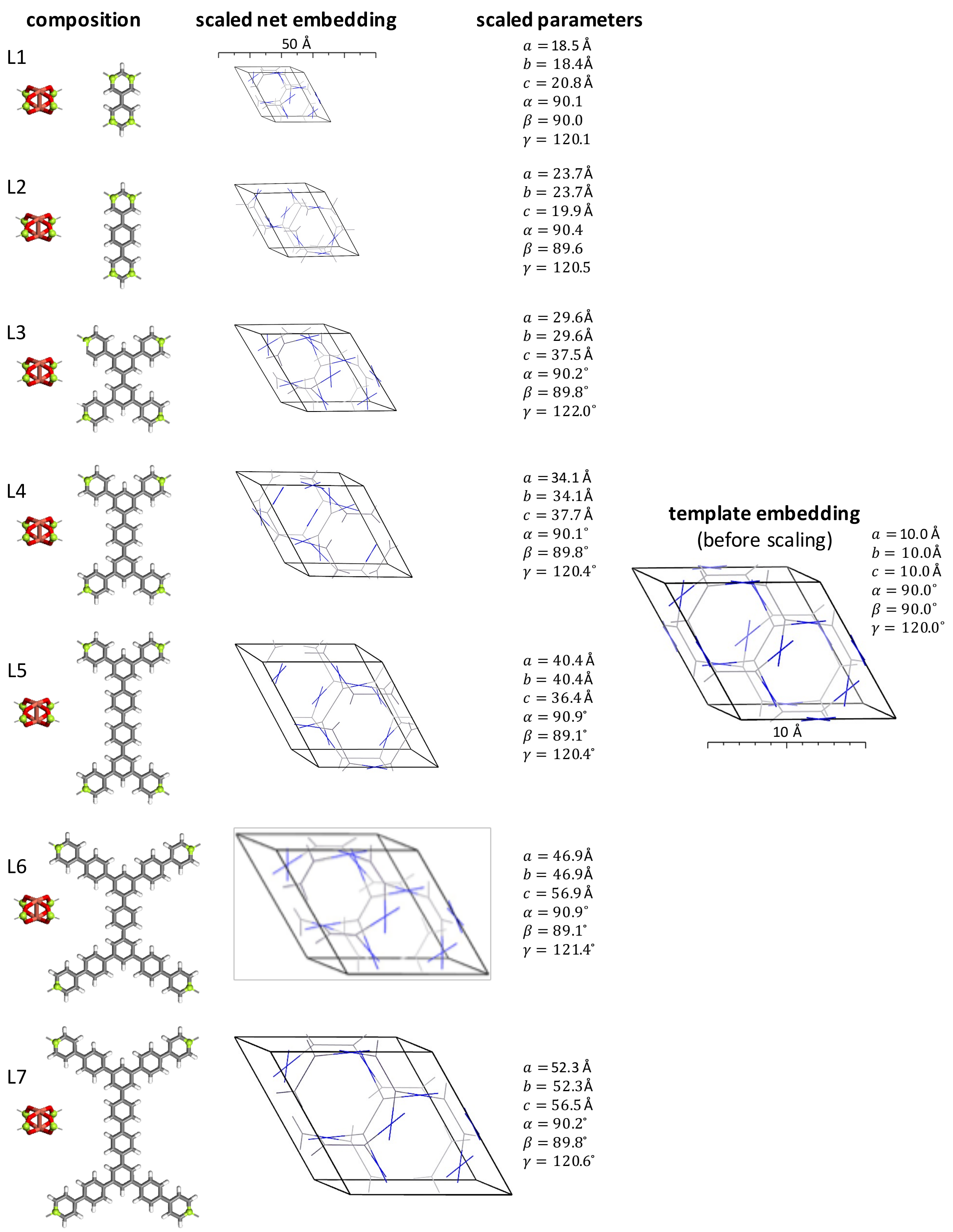

Figure S3. Example of how ToBaCCo anisotropically rescaled the same topological blueprint to build seven MOFs based on linkers of different size and aspect ratio. The original barycentric embedding of the **fog** topology (chosen as example) is shown to the right. Inorganic nodes and organic linkers used in MOF construction are shown to the left. The anisotropically rescaled net embeddings (all shown to scale) with the corresponding unit cell parameters are shown at the center. Green atoms represent connection sites in the building blocks. Connection sites form bonds between them.

■ Constructing topological blueprints

CIFs are commonly used to visualize molecular and atomic crystal structures, and contain at least three sections: i) the crystal unit cell parameters, ii) space group and symmetry operations, and iii) atomic fractional coordinates. The latter are accompanied by the corresponding label and element symbol for the atom for which the coordinates are defined for. However, it is also possible to add extra information such as the partial charges of the atom. Sometimes a fourth section containing connectivity information is added. Given the familiarity of the MOF community with CIFs, we decided to modify ToBaCCo so it can receive all information needed to construct MOFs using CIFs.

ToBaCCo now reads the topological blueprint (or template) from a CIF, which is written in P1 symmetry, even if the template has higher symmetry. Thus, the coordinates for all vertices are explicitly written in the CIF. ToBaCCo expects sets of vertices which are symmetrically equivalent in the barycentric embedding to be all labelled as the same element but different numerical index. For example, a set of symmetrically equivalent vertices could be labeled V1, V2, ..., V_N, and another set could be labeled Er1, Er2, ..., Er_M. The reason to use atomic elements as vertex labels is that i) this is what available software expect to read when it reads a CIF, ii) when using software to build CIFs this is what the software will allow the user to use through its GUI.

To keep formatting somewhat standardized, ToBaCCo ***expects the first five sets of symmetrically equivalent vertices to be labeled using the elements V, Er, Ti, Ce, and S, respectively, in that order***. The next sets of symmetry equivalent vertices should be labeled H, He, Li, Be…and continuing in increasing order of atomic number. So far, no more than 30 distinct labels are needed, as the maximum number of symmetry inequivalent vertex types in the RCSR is 29. Coordinates for net edges (the middle point between connected vertices) are to not be provided. Instead, net edges are recognized from the connectivity section of the CIF, leveraging the alternative definition of edge as a tuple of connected vertices (each edge correspond to a listed pair of connected vertices).

■ Understanding building block assignment to blueprint nodes and edges

A nodular building block is deemed compatible with a given vertex type in a template when the number of connection points in the building block matches the number of incident edges for the given vertex type. By default, ToBaCCo will build different MOFs for all possible permutations of building blocks based on their compatibility with the vertices in a template. Alternatively, the user can specify that only one MOF get built, in which the nodular building block that best fits each vertex in the barycentric version of the net will be used. Also, alternatively, a user that may have a specific MOF target in mind could provide ToBaCCo with a specific list (in a separate text file) containing the specific desired assignment of building block to each vertex type.

Since now ToBaCCo handles nets with more than one type of vertex, the assignment of connecting building blocks to be mapped onto edges is more complex than in the original code. ToBaCCo recognizes edge type by the types of vertices it connects. For instance, all edges connecting, say, V_N to Er_N vertices could be type 1, while those connecting V_N to V_N could be type 2. Edges assigned to be of the same type will all "host" the same connecting building block in each MOF. Note that in the barycentric embedding, edges of the same type are symmetrically equivalent. ToBaCCo now loops through all possible combinations of the provided edges i.e. and ordered list of edges for each iteration. In the special case that the template has only one type of edge, the edge assignment procedure is the same as in previous versions of ToBaCCo.

Alternatively, the user can specify that the edges are assigned as they appear in the edge library (i.e. in canonical alphanumeric order), allowing for targeted single MOF construction. ***The connection points in the building blocks CIFs are recognized by ToBaCCo, because they are labeled as X1, X2, X3, …***

■ On how template scaling is now performed in ToBaCCo.

Once ToBaCCo has made the assignment for the molecular building blocks to each vertex and edge type in the template, the template needs to be rescaled to fit the geometry of specific building blocks as well as possible. To quantify the fit, an objective function defined by the total deviation between the template and the building blockss is specified. To optimize the fit, the objective function is to be minimized, resulting in a rescaled template that overlaps the chosen building blockss as closely as possible. The voltage graph (or labeled quotient graph) representation of a crystallographic net makes defining this deviation relatively straightforward. However, let us first discuss what a voltage graph is.[1–3] To make the explanation more accessible, we often make reference to how the crystallographic **qtz** net was converted to a voltage graph (Figure S4).

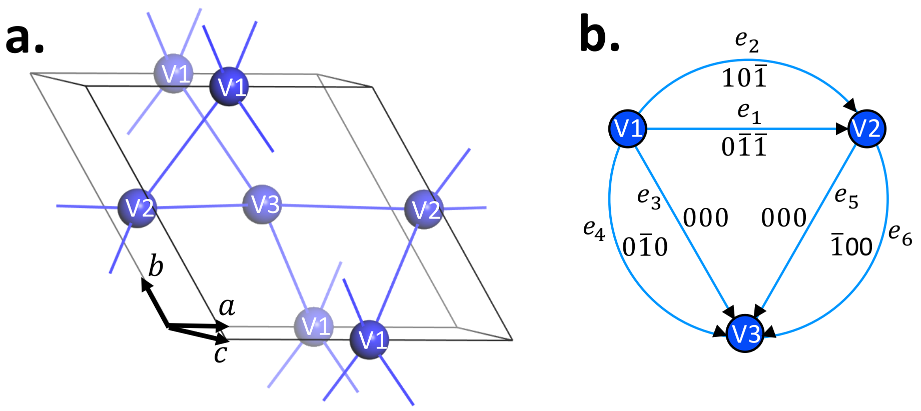

Figure S4. a) The **qtz** net unit cell, which has three types of vertices, in its barycentric embedding. c) The voltage graph with edges labelled e_i. The voltage graph vertices are labelled by the unit cell vertices which are in the quotient groups they represent.

__**Constructing a voltage graph.**__ To construct the **qtz** voltage graph, we take the unit cell of the **qtz** net in Euclidian space as the reference (Figure S4a). First, we note that V1 (or any V1 translational equivalent) is connected to four other vertices: i) two vertices that are translationally equivalent to V2, ii) V3, and iii) one vertex that is translationally equivalent to V3. A similar analysis can be made for V2 and V3. Then, to construct the voltage graph, we first define *quotient groups*, where the members of each group are translationally equivalent vertices. Each quotient group is represented by a vertex in the voltage graph, and each edge of the voltage graph (Figure S4b) represents a connection between a member of a quotient group and a member of another quotient group (or sometimes the same quotient group). For instance, for the **qtz** net, the V1 quotient group is the group constituted by the vertex V1 and all vertices translationally equivalent to V1.

Since all members of the V1 quotient group are connected to two equivalents of V2, there will be two edges (e_1 and e_2) connecting V1 and V2 in the voltage graph. To draw e_1 we say "V1 is connected to the vertex obtained by translating V2 by the vector (a, 0, -c)" and hence draw the edge e_1 pointing from V1 to V2, and then label the edge with the translation vector that relates V2 (in the unit cell) to the vertex that V1 is connected to. This translation vector is the "voltage" of the edge and it is conventionally represented by the vector coefficients with an overbar used to signify negatives. In this case, $(a, 0b, -c)$ is represented as $10\bar{1}$. Note that e_1 could have been drawn pointing from V2 to V1, in which case the translation vector labeling the edge would have opposite sign. The **qtz** voltage graph in Figure S4b was obtained by following this procedure for all vertices. Note that a net is uniquely represented by its voltage graph, i.e. no other net has the same voltage graph.[3] However, a net can have multiple voltage graphs, as the edge directions and vertex labels are not fixed. Voltage graphs are useful in creating embeddings of nets in Euclidean space, so to understand this representation better, let us discuss a few definitions first.

Relevant definitions from algebraic graph theory. A *vector space* consists of a group of vectors S and a field of scalars F such that the sum of any two vectors in S is also a vector in S and that the multiplication of any vector in S by any scalar in F is also a vector in S. The quintessential example of a vector space is $\mathbb{R}^3$, where vectors are 3-tuples of real numbers and scalars are real numbers. However, vector spaces can be more abstract, because, for instance, the vector addition operation does not have to be defined as the "traditional" addition we are used to when adding vectors in $\mathbb{R}^3$. Rather, *any* operation between two vectors in S that produces another vector in S can be defined as "addition." Moreover, the field of scalars F need not be infinite. For instance, the field could be constituted by just 1, -1, and 0.

The *basis* of a vector space S is defined as any set of linearly independent vectors (excluding the zero vector), where every vector in S can be formed as a linear combination of this set. For example, the vectors $(1,0,0)$, $(0,1,0)$, $(0,0,1)$ form a well-known basis for $\mathbb{R}^3$. Finally, the *dimension* of the vector space is defined as number of vectors required to complete a basis. With these standard definitions in mind, we now proceed to discuss some of the algebraic properties of voltage graphs.

The *edge space* of a voltage graph is defined as the vector space where the field of scalars is the set {-1, 0,1} and:

- Vectors are subsets of the edge set. For instance, for the **qtz** voltage graph, the edge set is $\{e_1, e_2, e_3, e_4, e_5, e_6\}$, and some subsets of the edge set are $\{e_1, e_2\}$, $\{e_3, e_4\}$, $\{e_5\}$, $\{e_2, e_4, e_6\}$, and other such combinations. Accordingly, a convenient basis for any edge space (heretofore called the canonical basis) is the set of edges of the graph (represented as subsets with one element), meaning the edge space of a graph has dimension N_e, where N_e is the number of edges. For the **qtz** voltage graph, this basis is $\{\{e_1\}, \{e_2\}, \{e_3\}, \{e_4\}, \{e_5\}, \{e_6\}\}$.

- The "addition" operation between edge space vectors e_1 and e_2 is defined as $e_1 \triangle e_2$. Where the $\triangle$ symbol represents the "symmetric difference" operation, which returns all the elements that are in one set but not in the other. For instance, for the **qtz** graph, $\{e_1, e_3\} \triangle \{e_2, e_3\}$ is equal to $\{e_1, e_2\}$, which is also a member of the edge space.

The *cycle space* of a voltage graph is a subspace of the edge space whose vectors are cycles, where each cycle is represented as the subset of edges that comprise the cycle. A *cycle* is a path that traverses from vertex to vertex along edges, and starts and ends on the same vertex. For instance, for the **qtz** voltage graph a cycle could be $\{e_1, e_5, e_4\}$ and another cycle could be $\{e_1, e_2\}$. Note that, consistent with the definition of vector space, the "symmetric difference" operation between these two cycles produces another cycle: $\{e_2, e_5, e_4\}$. A basis of the cycle space is called the *cycle* *basis*, and has $N_e - N_v + 1$ elements (for a connected graph),[1,4] where N_v is the number of vertices. For instance, a basis for the **qtz** voltage graph has four (6 - 3 + 1) elements. A basis could be $\{\{e_1, e_2\}, \{e_3, e_4\}, \{e_5, e_6\}, \{e_2, e_4, e_6\}\}$.

The *co-boundary* of a subset of vertices of a graph is the set of edges connecting vertices in the subset to vertices not in the subset.[1] For instance, the co-boundary of the subset of vertices{V1,V2} of the **qtz** voltage graph is the set $\{e_1, e_2, e_5, e_6\}$. The *co-cycle* space of a graph is a subspace of the edge-space whose vectors are co-boundaries.[1] The co-cycle space of a graph has dimension $N_v - 1$, and the set of co-boundaries of the first $N_v - 1$ *individual* vertices in a graph can be a basis for the co-cycle space.[1] For instance, for the **qtz** voltage graph a basis for the co-cycle space could be $\{\{e_1, e_2, e_3, e_4\},\{e_1, e_2, e_5, e_6\}\}$, which are the co-boundaries of V1 and V2. Notice that the sum of $N_e - N_v + 1$ (the dimension of the cycle space) and $N_v - 1$ (the dimension of the co-cycle space) is N_e (the dimension of the edge space). The cycle and co-cycles bases are both comprised of linearly independent vectors, furthermore, since the edges incident on a subset of vertices cannot form a cycle, cycles and co-boundaries are inherently linearly independent. Thus, since the size of the cycle basis combined with the co-cycle basis is also the dimension of the edge space, ***the cycle-co-cycle basis combination is an alternative basis of the edge space***.

Constructing Euclidean embeddings using the cycle and co-cycle bases. A crystallographic net can be embedded (mapped) into $\mathbb{R}^3$ by assigning a 3-component vector to each edge of the voltage graph. One only needs to define a matrix operator that maps the set of edges (i.e. the canonical basis of the edge space) to a set of 3-component vectors. Such a representation would already, in principle, allow the geometric deviation between the embedding and building blocks to be calculated, hence allowing an objective function of this deviation to be defined. However, an equivalent basis, that of the cycle basis combined with the co-cycle basis (from now on referred to as the cycle-co-cycle basis) of the voltage graph, can be mapped to a set of 3-vectors using so-called lattice and co-lattice vectors. This later mapping provides a more convenient definition for the problem of scaling a crystallographic net, as will subsequently be shown.

A *lattice vector* is a translation vector that "transforms" points in the crystallographic unit cell to an equivalent point in a repeat unit. Accordingly, the sum of voltages (*net voltage*) in a cycle of a voltage graph must be a lattice vector. The net voltage must correspond to a lattice vector because a cycle in a voltage graph defines a path from a point in the net unit cell, to a translationally equivalent point in a repeat unit (recalling that each node in the voltage graph is a group of equivalent vertices). A *co-lattice* vector is similar to a lattice vector, except defined on a co-boundary and can be thought of as the sum of the 3-vectors corresponding to the edges in the co-boundary for an embedding in Euclidean space.[1] Keep in mind that lattice (co-lattice) vectors are the Euclidian space equivalents of cycles (co-boundaries), which means they can be added just like vectors in $\mathbb{R}^3$ can. Then, because the cycle-co-cycle basis forms a basis for the edge space, the corresponding lattice and co-lattice can be used to define a function that maps the cycle-co-cycle basis to the edge vectors of an embedding in Euclidean space. Using this mapping, a representation of a net in Cartesian coordinates can be constructed from the cycles and co-boundaries of the voltage graph and the corresponding lattice and co-lattice vectors.

If a crystallographic net can be embedded directly into Euclidean space by mapping its edges to 3-vectors, then ***why bother with cycles and co-boundaries?*** The reason is to exploit the fact that the co-lattice vector of all vertices in any barycentric embedding must be equal to the zero vector, because in barycentric embeddings all vertices are the center of mass of their neighboring vertices. Therefore, a non-zero co-lattice vector directly quantifies the magnitude and direction of a vertex deviation from its original barycentric placement. Consequently, the co-lattice vectors of a net embedding can be used to quantify its total deviation from its barycentric embedding. Since we are concerned with finding lower symmetry embeddings that match building block geometries, the lattice vectors are extremely useful geometric measures, because they can be used as variables in an objective function that allows the fractional coordinates of vertices to deviate from their barycentric positions to better fit building block geometry. However, since a voltage-graph co-cycle space only forms an $N_v - 1$ -dimensional subspace of the edge space, a cycle basis must be used together with co-cycle basis to form a complete edge space basis and, thus, a complete representation of the edge geometry of the net of interest. To clarify the above concepts, the entire process of constructing an objective function from the voltage graph representation of the **qtz** net is shown next, and the corresponding details in `ToBaCCo` explained.

Detailed description of the `ToBaCCo` scaling algorithm using the qtz net as example. In `ToBaCCo`, vectors of edges, such as cycles and co-boundaries, are represented as ordered N_e-tuples (recall that we have defined N_e as the number of edges in a graph). For example, if edges e_i, e_j, and e_k form a cycle in some voltage graph, the tuple representing that cycle would have a 1 at positions i, j, and k and 0 elsewhere. If any edge is traversed backwards in the cycle, a negative 1 is used at that edge position. The exact numbering (or ordering) of the edges is irrelevant as long as it is kept consistent for every cycle. Since the **qtz** unit cell has six edges, cycles and co-boundaries in the **qtz** voltage graph are represented as 6-tuples. We construct the cycle-co-cycle basis by first constructing the cycle and co-cycles bases separately and then combining them. In `ToBaCCo`, the cycle basis is constructed by first finding the spanning tree of the voltage graph of interest. The *spanning tree* of a graph G is the subgraph which connects all the nodes of G with the minimum possible number of edges and that contains *no cycles*.[4] An element of the cycle basis is then found by adding an edge not present in the spanning tree back into spanning tree to form a new subgraph, which conveniently has exactly one cycle (the new cycle-basis member). Repeating this process for each edge not present in the spanning tree yields a cycle basis.[4] For the **qtz** voltage graph, however, a cycle basis can be found easily by inspection. `ToBaCCo` gives

$$\{(1,-1,0,0,0,0),(0,0,-1,1,0,0),(1,0,0,-1,1,0),(1,0,0,-1,0,1)\}$$

for the edge ordering shown in Figure S4. Note that there are four cycles in the basis, as we expect. Next, a co-cycle basis must be constructed. In `ToBaCCo`, a co-cycle space is built with the co-boundaries of the first $N_v - 1$ vertices (although the co-boundaries of any $N_v - 1$ vertices would work just as well). For **qtz** `ToBaCCo` gives the co-cycle basis

$$\{(1,1,1,1,0,0),(-1,-1,0,0,1,1)\}$$

with, vitally, the same edge ordering as was used for constructing the cycle basis. Next, we calculate the lattice vectors corresponding to the cycles in the cycle basis (i.e. the cycle net voltage):

$$\{\bar{1}\bar{1}0, 0\bar{1}0, 00\bar{1}, \bar{1}0\bar{1}\}.$$

Then, the co-lattice vectors are defined as variables (two 3-vectors, or six variables total):

$$\{x_1 x_2 x_3,\ \ x_4 x_5 x_6\}.$$

In the barycentric embedding, all the x_i are 0. Using the cycle-co-cycle basis a matrix B^* is built:

$$B^* = \begin{pmatrix} 1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 1 & 0 & 0 \\ 1 & 0 & 0 & -1 & 1 & 0 \\ 1 & 0 & 0 & -1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 \\ -1 & -1 & 0 & 0 & 1 & 1 \end{pmatrix},$$

where the first $N_e - N_v + 1$ (4) rows are the cycle basis vectors and the next $N_v - 1$ (2) rows are the co-cycle basis vectors.[1] The inverse of B^* is then used to map the lattice and co-lattice vectors to a Euclidean representation of the voltage graph edges. The lattice and co-lattice vectors are encoded in the matrix α.

$$\alpha(x_1, x_2, x_3, x_4, x_5, x_6) = \begin{pmatrix} -1 & -1 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \\ -1 & 0 & -1 \\ x_1 & x_2 & x_3 \\ x_4 & x_5 & x_6 \end{pmatrix},$$

where the first $N_e - N_v + 1(4)$ rows are the lattice vectors corresponding to the first $N_e - N_v + 1$ rows of B^* (the exact ordering does not matter so long as it is consistent with B^*), and the next $N_v - 1$ (3) rows are the co-lattice vectors corresponding to the last $N_v - 1$ rows of B^*. Put more succinctly, the matrix

$$\Omega = B^{*-1}\alpha(0,0,0,0,0,0) = \begin{pmatrix} -1/2 & -1/2 & -1/3 \\ 1/2 & 1/2 & -1/3 \\ 0 & 1/2 & 1/3 \\ 0 & -1/2 & 1/3 \\ 1/2 & 0 & -1/3 \\ -1/2 & 0 & -1/3 \end{pmatrix}$$

gives the vectors, in fractional coordinates, corresponding to edges of the *barycentric* embedding (all $x_i = 0$) of **qtz**.[1] The full reason why this is the correct mapping between the cycle-co-cycle basis and the edge vectors is beyond the scope of this contribution, but is described in detail by Eon.[1] Briefly, B^* maps the edge vectors in Ω to their lattice/co-lattice representation in α (being the change of basis matrix between the canonical edge space basis and the cycle-co-cylce basis), and, thus, B^{*-1} performs the inverse mapping.

The matrix Ω, then, specifies an embedding in fractional coordinates that is a function of the co-lattice vectors. However, we are also interested in how the template unit cell parameters can be used to adjust vertex geometry to fit building blocks. Therefore, we compare net geometry to building block geometry in Cartesian coordinates, as conversion between fractional and Cartesian coordinates requires a matrix operator Z, that is built using all six lattice parameters. More precisely, we compare the pairwise dot-products (calculated in Cartesian coordinates) of the net edges to the connection site vector angles and lengths of the assigned building blocks, as suggested by Boyd and Woo.[2] Using dot products instead of coordinates negates the need for the many coordinate frame shifts and rotations needed to directly compare vertex and building block coordinates.[2] An $N_e \times N_e$ matrix of the pairwise edge dot-products in Cartesian coordinates (where the ij entry corresponds to the dot product between e_i and e_j) is formed from the product $P = \Omega Z \Omega^T$ where

$$Z = \begin{pmatrix} a^2 & (ab)\cos(\gamma) & (ac)\cos(\beta) \\ (ab)\cos(\gamma) & b^2 & (bc)\cos(\alpha) \\ (ac)\cos(\beta) & (bc)\cos(\alpha) & c^2 \end{pmatrix},$$

and $a, b, c, \alpha, \beta, \gamma$ are the template unit cell parameters[1,2] (the product $\Omega\Omega^T$ would give the pairwise dot products in fractional coordinates). Thus, for **qtz**:

$$P(a,b,c,\alpha,\beta,\gamma,x_1,x_2,x_3,x_4,x_5,x_6) \\ = B^{*-1}\alpha(x_1,x_2,x_3,x_4,x_5,x_6)\begin{pmatrix} a^2 & (ab)\cos(\gamma) & (ac)\cos(\beta) \\ (ab)\cos(\gamma) & b^2 & (bc)\cos(\alpha) \\ (ac)\cos(\beta) & (bc)\cos(\alpha) & c^2 \end{pmatrix}\left(B^{*-1}\alpha(x_1,x_2,x_3,x_4,x_5,x_6)\right)^T.$$

The entries in P can then be compared to the corresponding building block dot products. This requires first assigning an building block to each edge and vertex in the voltage graph, as described in Section S1.4. In `ToBaCCo,` this assignment is then translated into a direction and length for each edge which must be matched in the final scaled template. That is, if vertices V1 and V2 in the **qtz** net are assigned a nodular building block with distance d_1 between its center-of-mass and connection sites, and the edge between them is assigned a linker of length d_2, edges e_1 and e_2 must have length $d_1 + d_2 + 2d_3$, where d_3 is the desired bond length between nodular and connecting building blocks. In addition, the angle between e_1 and e_2 must match the angle between the node building block connection site vectors assigned to e_1 and e_2, respectively. Connection site vectors are assigned to edges by translating and rotating normalized node building blocks into the position of their assigned vertex *before any scaling occurs*, each connection site vector is then assigned the edge it is nearest after this preliminary placement. Assigning edges in this way ensures that the building block and vertex geometries are aligned as closely as possible before scaling begins. After the matrix P is constructed and the all edges assigned their desired lengths and directions, the final objective function can be defined.

For each nodular building block we have the terms

$$D_{ij} = L_i L_j \cos(\theta_{ij})$$

where L_i is the required distance between the assigned vertex and the vertex connected to it via e_i, L_j is the analogous distance for e_j, and θ_{ij} is the required angle between e_i and e_j, all as enforced by the linker/node building block geometry. These terms allow the objective function to be defined (for **qtz**) as

$$O(a,b,c,\alpha,\beta,\gamma,x_1,x_2,x_3,x_4,x_5,x_6) = \sum_{i=1}^{N_e}\sum_{j=1}^{N_e}(D_{ij} - P_{ij})^2 \,,$$

where P_{ij} is the ij entry in P. In general, O is a function of the unit cell parameters and $3(N_v - 1)$ co-lattice vector variables (rather than the six shown above). In `ToBaCCo`, this objective function is minimized using a bounded L-BFGS-B algorithm.[5,6] The variable starting values are taken to be those of the barycentric embedding.

The scaling method presented here and introduced to `ToBaCCo` follows the spirit of the method presented by Boyd and Woo.[2] However, these authors "augment" their voltage graphs before computing the pairwise edge dot-products by adding several "artificial" 2-connected vertices between the actual vertices. Since each new vertex increases the number of co-lattice vectors by one (see ref. 2 for more details), this has the effect of adding more degrees of freedom in α and thus to the objective function. Their rationale behind augmentation is that the extra vertices can help better accommodate building block geometry, especially in cases where the initial net vertex geometry is markedly different with respect to the building block geometry.

We briefly considered the above approach, by adding five extra 2-connected vertices between all voltage graph vertices. However, we found that this augmentation procedure often had the effect of creating undesirable angles between nodular and connecting building blocks while leaving the template unit cell parameters essentially unchanged during scaling. The reason this happened is that, with the extra vertices, there are multiples ways to minimize the objective function. As it turns out, the rescaling algorithm often prefers to minimize the "deviation" between a vertex and the corresponding building block geometry by unphysically "bending" the edges around vertex in question, instead of by altering the template unit cell parameters (see Figure S5). Although unphysical bending at connection sites could potentially be removed during geometry optimization, in our experience, the relaxation process can be significantly influenced by starting geometries which have large local deviations from equilibrium. Therefore, in our modification of `ToBaCCo`, we decided that nets should not be augmented (i.e. should not have extra vertices added) as to instead force the unit cell parameters to expand and contract more to suit the building block geometry.

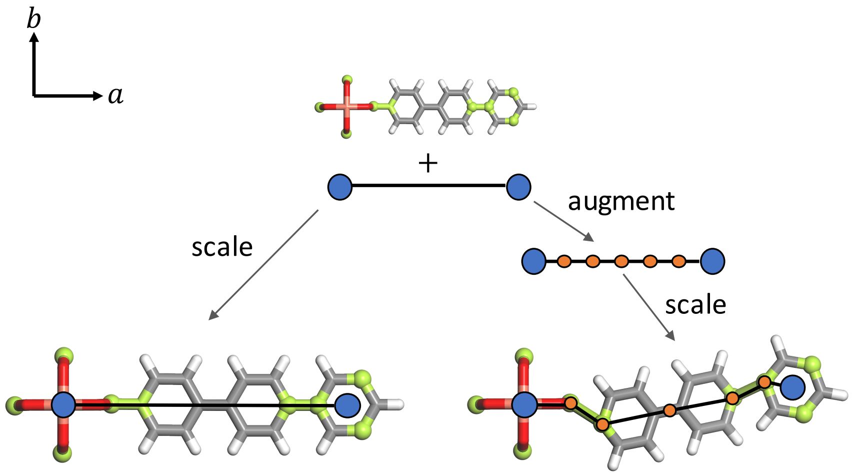

Figure S5. Scaling without augmentation (left branch) and with augmentation (right branch). In both cases the net edge length(s) and building block geometry match well, contribution a low value of the objective function, but the change in the lattice constant *a* is less for the augmented case, which is compensated for by the bending around node-to-linker connection sites (shortening the distance between the original vertices).

■ How ToBaCCo places the building blocks onto the rescaled net.

Once the template is scaled, the building blocks are translated into their fractional positions in the scaled unit cell and rotated to maximally overlap with their assigned edges. These transformations are accomplished using a script directly from the Biopython project,[7] which finds the translation vector and rotation matrices which minimize the root-mean-squared-deviation (RMSD) between two arrays of vectors.

■ Programming details and downloading the code.

`ToBaCCo` is written in `Python 2.7`.[8] The `NumPy` package is used for numerical calculations and matrix operations. The `SciPy` package is used to implement the L-BFGS-B optimization algorithm. `NetworkX` is used to build voltage graphs, spanning trees, and discover cycles for cycle-basis construction. The code can be obtained by emailing the authors or from the GitHub repository located at https://github.com/tobacco-mofs (look for ToBaCCo 3.0 version).

Section S2. `ToBaCCo` start-up instructions

■ Installation

The first step towards running `ToBaCCo` is installing `Python 2.7` with the required packages. This is most easily accomplished by installing the `Anaconda` distribution of `Python 2.7` (which includes all the packages required by `ToBaCCo` by default). Installers for Windows, macOS, and Linux can be found at

https://www.anaconda.com/download/#windows
https://www.anaconda.com/download/#macos
https://www.anaconda.com/download/#linux

respectively. Next, to obtain the `ToBaCCo` code visit https://github.com/tobacco-mofs/tobacco_3.0 and select the "clone or download" box on the right side of the page. This will download the code as a zip file that you can uncompress in your computer. Alternatively, MacOS and Linux users can type

```
git clone https://github.com/tobacco-mofs/tobacco_3.0.git
```

into a terminal window to download the code into the current folder (directory). As `ToBaCCo` is written in `Python` no further installation/compilation steps are required after obtaining the code from GitHub.

■ Organization of ToBaCCo files and folders

When you uncompress the downloaded zip, the *tobacco_3.0* folder will be released. You will work in this folder to automatedly construct porous crystals. The folder contains the actual ToBaCCo code files in `Python` (such as the main file *tobacco.py* and module such as *scale.py*, *place_bbs.py,* and so forth) and five other folders (subdirectories): *templates*, *nodes*, *edges*, *output_cifs*, and *check_cifs*. When the *tobacco.py* file is run, it expects to find these folders (Figure S6).

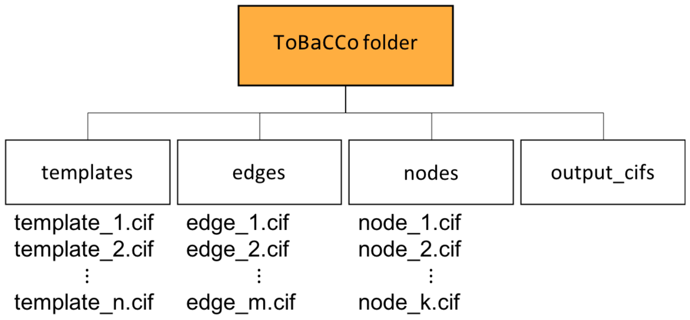

Figure S6. The directory tree of `ToBaCCo` with arbitrary inputs shown

templates: This is the folder where the CIFs for the topological blueprints should be placed. `ToBaCCo` will attempt to use all template files inside this folder during each run, so *i)* only files formatted as templates should be present, *ii)* only templates that you are interested in using should be in the folder (otherwise the code will keep running longer than necessary). As building and validating templates requires specialized knowledge not necessarily held by all potential users, we continuously update the ToBaCCo git repository with new templates (pre-placed in the "templates" directory) as we construct and validate them. This allows any user to start MOF synthesis immediately using the already included templates.

nodes: This is the folder where the CIFs for the nodular building blocks should be placed. All node CIFs will be assessed by ToBaCCo to see if there are viable to be mapped onto the vertices of the template that the code is using at the time. If ToBaCCo does not find suitable nodular building blocks for a given template, the code will skip making structures for that template.

***edges**:* This is the folder where the CIFs for the edge (connecting) building blocks should be placed.

output_cifs: This is the folder where ToBaCCo places the structures computationally synthesized by the code. If the folder is not present, the code will create it at the moment of running.

■ Running ToBaCCo

To run the code, first make sure that all the desired templates, connecting building blocks and nodular building blocks are placed in their respective folders. The crystal construction process will start by running the `tobacco.py` script in an Anaconda prompt (installed with Anaconda) in Windows, or in a terminal window in MacOS or Linux. The running command should be run while inside the *`tobacco_3.0`* directory downloaded from GitHub.

The ToBaCCo standard output gives information about the building block assignment, unit cell scaling, bond formation, and charges (if applicable). Information on more advanced usage and on user defined parameters can be found in the manual included with the ToBaCCo distribution.

■ MOF construction example (fog MOFs studied in this work)

In the following paragraphs, we will guide the readers through the construction of the seven **fog** MOFs considered in this work.

Preparing the input files. You will need to have the building blocks and templates in their respective folders. The edge building blocks (*L_12.cif*, *L_24.cif*, and *ntn_edge.cif*) and node building blocks (*sym_5_mc_2.cif* and *sym_3_on_2.cif*) are already placed in the respective folders in the *tobacco_3.0* folder that you uncompressed. If you are going to use your own building blocks in your own work, just make sure they follow the same CIF format as the example building blocks provided, with the connection points labeled as X1, X2, X3…

The *templates* folder within the *tobacco_3.0* folder that you uncompressed should contain over 70 topological blueprints. For this exercise, make sure that only the file *fog.cif* is within the *templates* folder (i.e. remove the all the other template CIFs included with ToBaCCo).

Overall, your folders and file distribution should look as illustrated in Figure S7 (top).

Running the code Once the template, node, and edge files are in the correct location the code is run with the command

```
python tobacco.py
```

in an `Anaconda` prompt (Windows) or a terminal window (MacOS and Linux). The MOF construction process will continue until all possible topology/node/edge combinations are used. In this case nine MOFs will be output (three edge building blocks for two edge types, together with one topology and one possible node assignment). Once the code finish running, your folders and file distribution should look as illustrated in Figure S7 (bottom). The two MOFs with the L_24.cif/NTN.cif edge combination were not considered here. Once these are deleted, the remaining seven structures are the (unoptimized) **fog** MOFs used for this study. Obtained MOFs, as well as provided building blocks and templates can be examined visually with any CIF visualizer of your preference.

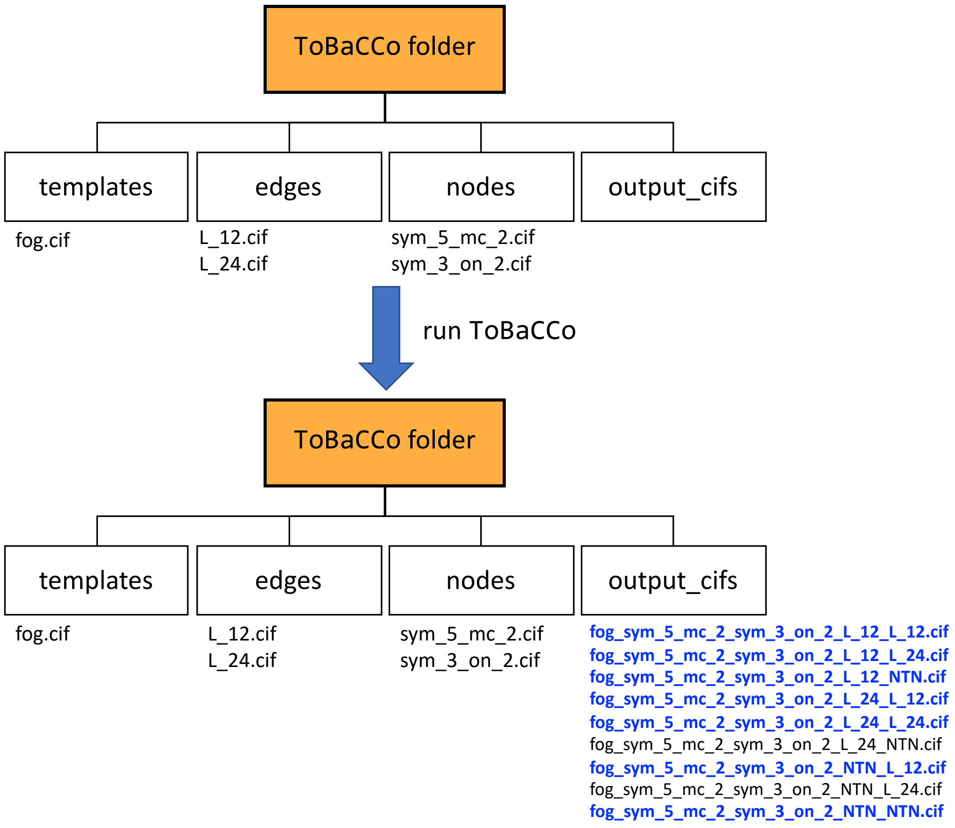
ToBaCCo folder
templates
edges
nodes
output_cifs
fog.cif
L_12.cif
L_24.cif
sym_5_mc_2.cif
sym_3_on_2.cif
run ToBaCCo
ToBaCCo folder
templates
edges
nodes
output_cifs
fog.cif
L_12.cif
L_24.cif
sym_5_mc_2.cif
sym_3_on_2.cif
fog_sym_5_mc_2_sym_3_on_2_L_12_L_12.cif
fog_sym_5_mc_2_sym_3_on_2_L_12_L_24.cif
fog_sym_5_mc_2_sym_3_on_2_L_12_NTN.cif
fog_sym_5_mc_2_sym_3_on_2_L_24_L_12.cif
fog_sym_5_mc_2_sym_3_on_2_L_24_L_24.cif
fog_sym_5_mc_2_sym_3_on_2_L_24_NTN.cif
fog_sym_5_mc_2_sym_3_on_2_NTN_L_12.cif
fog_sym_5_mc_2_sym_3_on_2_NTN_L_24.cif
fog_sym_5_mc_2_sym_3_on_2_NTN_NTN.cif

Figure S7. The directory tree of `ToBaCCo` with inputs necessary to build the **fog** MOFs used in this study before (top) and after (bottom) running the program. Not all template/node/edge combinations were considered, the output CIFs used are highlighted in blue.

Section S3. Comparison of computational and experimental MOFs

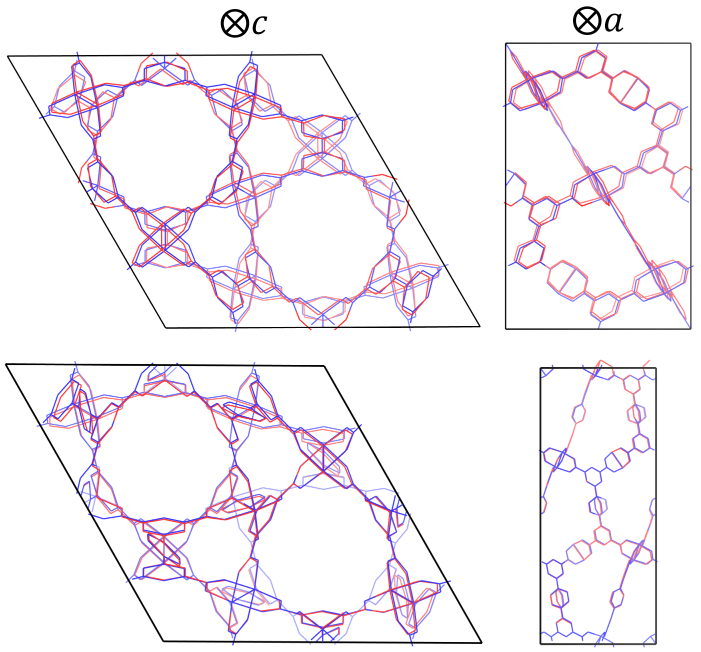

Figure S8. Overlap of the experimental structures (blue) and computational structures (red) of NOTT-100/MOF-505[9,10] (top) and NOTT-101[9] (bottom). Note that the computational NOTT-100/MOF-505 analog shown here is the exact mirror image of the structure used in our simulations, the mirror image was used to match the experimental structure. The experimental structures of NOTT-100/MOF-505 and NOTT-101 are from the Cambridge Structural Database, with identifiers LASYOU and CESFOW, respectively. Extra oxygen atoms coordinating the Cu paddlewheels and any crystallographic disorder were removed from the experimental structures before comparison.

Section S4. Calculation of MOF crystal mechanical properties

■ Hooke's law and Voigt notation

As stated in the main text Hooke's law in tensorial form is given by Equation 1, reproduced below for convenience:

$$\sigma_{ij} = \sum_{kl} c_{ijkl}\epsilon_{kl} \qquad (1)$$

where σ_{ij} and ϵ_{kl} are components of the stress and strain tensor, respectively. The indices i, j, k, l range between three directions (see Figure S8).

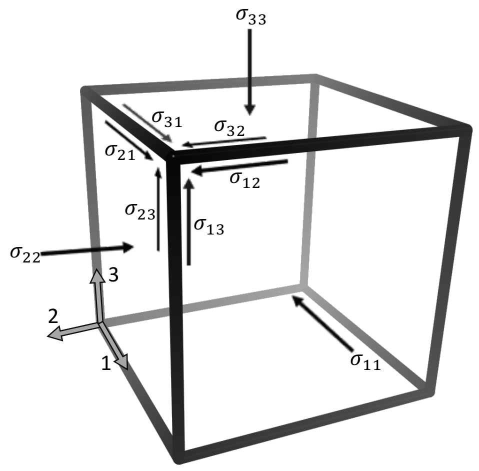

Figure S9. Components of the stress tensor in each direction.

In general, the stress tensor can be written as a 3 × 3 symmetric matrix:

$$\begin{pmatrix} \sigma_{11} & \sigma_{12} & \sigma_{13} \\ \sigma_{21} & \sigma_{22} & \sigma_{23} \\ \sigma_{31} & \sigma_{32} & \sigma_{33} \end{pmatrix};$$

meaning there are six unique elements. The strain tensor is written in analogously:

$$\begin{pmatrix} \epsilon_{11} & \epsilon_{12} & \epsilon_{13} \\ \epsilon_{21} & \epsilon_{22} & \epsilon_{23} \\ \epsilon_{31} & \epsilon_{32} & \epsilon_{33} \end{pmatrix},$$

and is again symmetric (i.e. it has six unique elements). In Voigt notation (where indices are mapped thus: 11 → 1, 22 → 2, 33 → 3, 23 → 4, 13 → 5, and 12 → 6), the stress and strain tensors are written as 6-vectors:

$$(\sigma_1 \quad \sigma_2 \quad \sigma_3 \quad \sigma_4 \quad \sigma_5 \quad \sigma_6)^T \ , \ (\epsilon_1 \quad \epsilon_2 \quad \epsilon_3 \quad \epsilon_4 \quad \epsilon_5 \quad \epsilon_6)^T,$$

and the elasticity tensor (C) can, thus, be written as a 6×6 matrix:

$$C = \begin{pmatrix} c_{11} & c_{12} & c_{13} & c_{14} & c_{15} & c_{16} \\ c_{21} & c_{22} & c_{23} & c_{24} & c_{25} & c_{26} \\ c_{31} & c_{32} & c_{33} & c_{34} & c_{35} & c_{36} \\ c_{41} & c_{42} & c_{43} & c_{44} & c_{45} & c_{46} \\ c_{51} & c_{52} & c_{53} & c_{54} & c_{55} & c_{56} \\ c_{61} & c_{62} & c_{63} & c_{64} & c_{65} & c_{66} \end{pmatrix}$$

with each entry is calculated via equation 2 in the main text, reproduced below for convenience

$$c_{ij} = \frac{\partial \sigma_i}{\partial \epsilon_j} = \frac{1}{V}\frac{\partial^2 E}{\partial \epsilon_i \partial \epsilon_j} \qquad (2).$$

■ Calculating elastic constants and bulk moduli

To compute the elastic constants (c_{ij}) for a crystal using molecular mechanics, Equation 2 must be evaluated numerically as $\Delta\sigma_i/\Delta\epsilon_j$.This is most easily accomplished by applying a small deformation (strain) in direction j and calculated the resulting change in the stress for each value of i (one through six). After each deformation, the atom positions are relaxed (keeping the unit cell parameters fixed) using a conjugate gradient algorithm (with energy and force convergence criteria as described in the main text), after which all six components of the stress tensor are calculated. Thus, for each value of j, a row of C can be calculated. In practice, both and positive and a negative deformation are applied, i.e. both expansion and contraction in direction j, or forward and reverse shear, giving two possible values of c_{ij} (one for the positive deformation and one for the negative deformation), these two values are then averaged to produce the final elastic constant. An example LAMMPS script which accomplishes this task for MOF Cu-BTC is given in the supplementary materials.

We calculated each bulk modulus by averaging the Voigt bulk modulus:

$$9K_V = (c_{11} + c_{22} + c_{33}) + 2(c_{12} + c_{23} + c_{31})$$

and the Reuss bulk modulus:

$$\frac{1}{K_R} = (s_{11} + s_{22} + s_{33}) + 2(s_{12} + s_{23} + s_{31})$$

where the s_{ij} are elements of the matrix S, which is the inverse of C.[11] K_V and K_R are the upper and lower bounds of the true bulk modulus, meaning their average is a more accurate estimate of K than either individual value.[11]

Section S5. Topological dependence of MOF textural properties

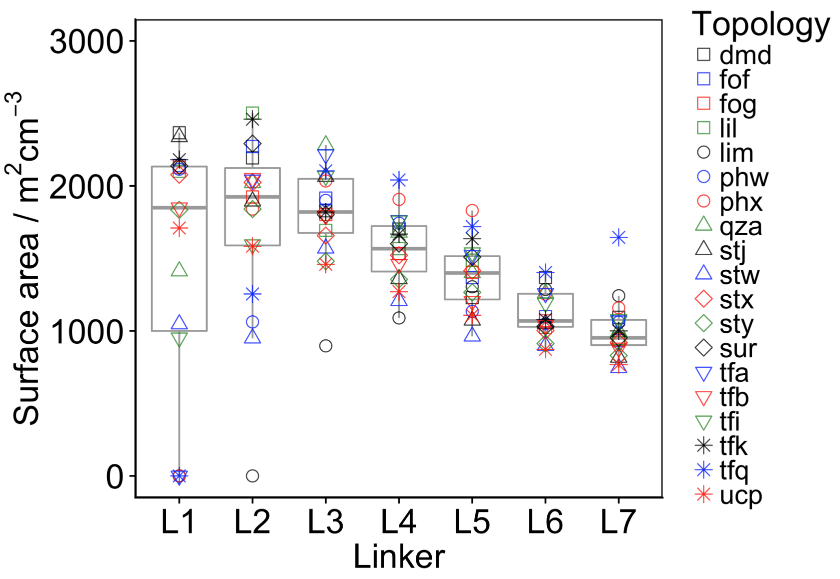

Figure S10. Volumetric surface area (calculated with a 1.82 Å radius probe) versus linker type each of the 126 MOFs is shown as a point with color/shape corresponding the legend shown. Box and whisker plots are shown behind the individual plots in grey, showing how the median and spread of surface area change with linker size.

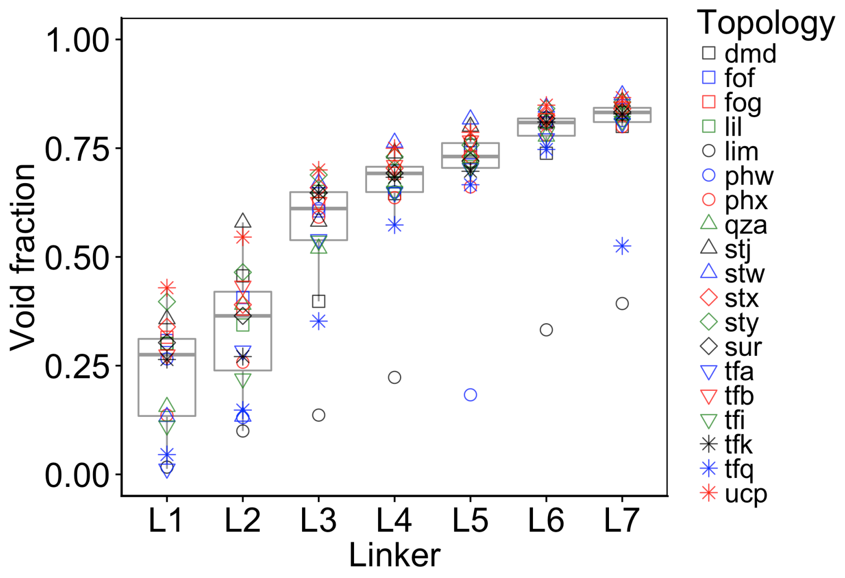

Figure S11. Helium void fraction (calculated with a 1.2 Å probe) versus linker type each of the 126 MOFs is shown as a point with color/shape corresponding the legend shown. Box and whisker plots are shown behind the individual plots in grey, showing how the median and spread of void fraction change with linker size.

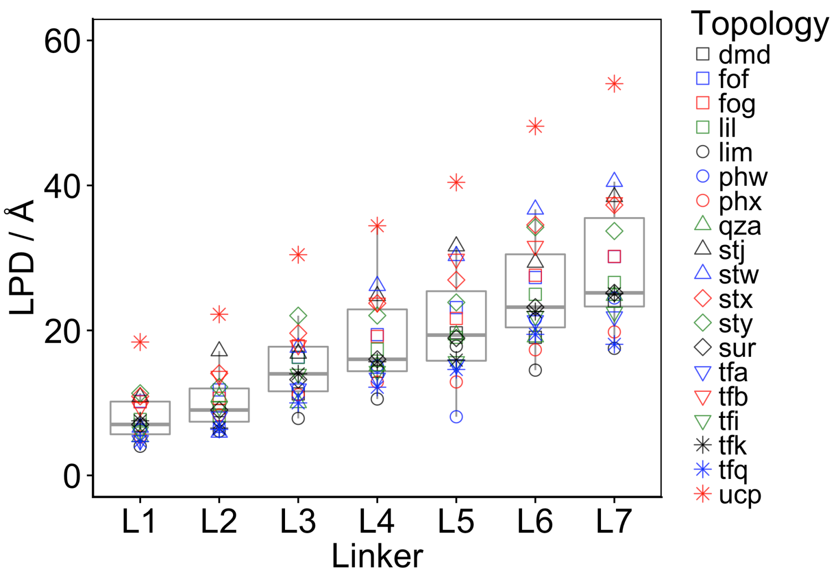

Figure S12. Largest pore diameter (LPD) versus linker type each of the 126 MOFs is shown as a point with color/shape corresponding the legend shown. Box and whisker plots are shown behind the individual plots in grey, showing how the median and spread of LPD change with linker size.

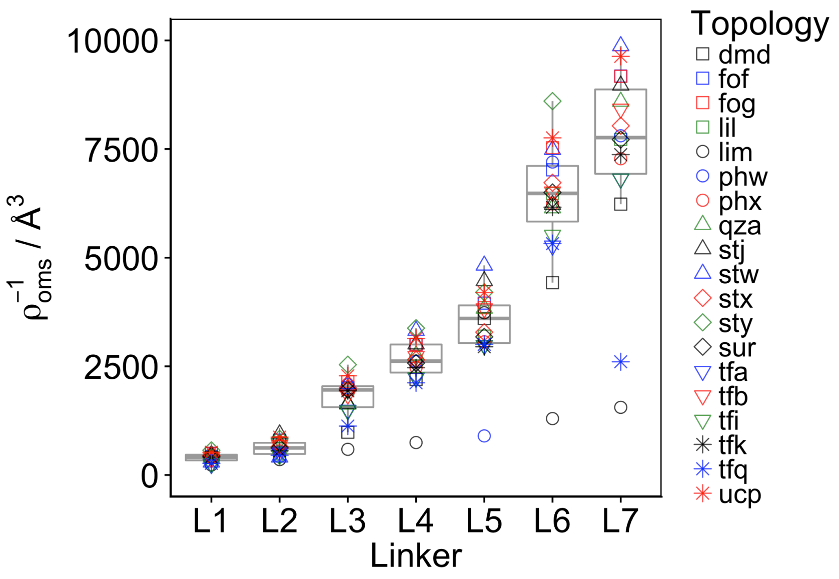

Figure S13. Inverse Cu open metal site density (ρ_{oms}^{-1}) verses linker type each of the 126 MOFs is shown as a point with color/shape corresponding the legend shown. Box and whisker plots are shown behind the individual plots in grey, showing how the median and spread of ρ_{oms}^{-1} change with linker size.

Section S6. Ethene selectivity

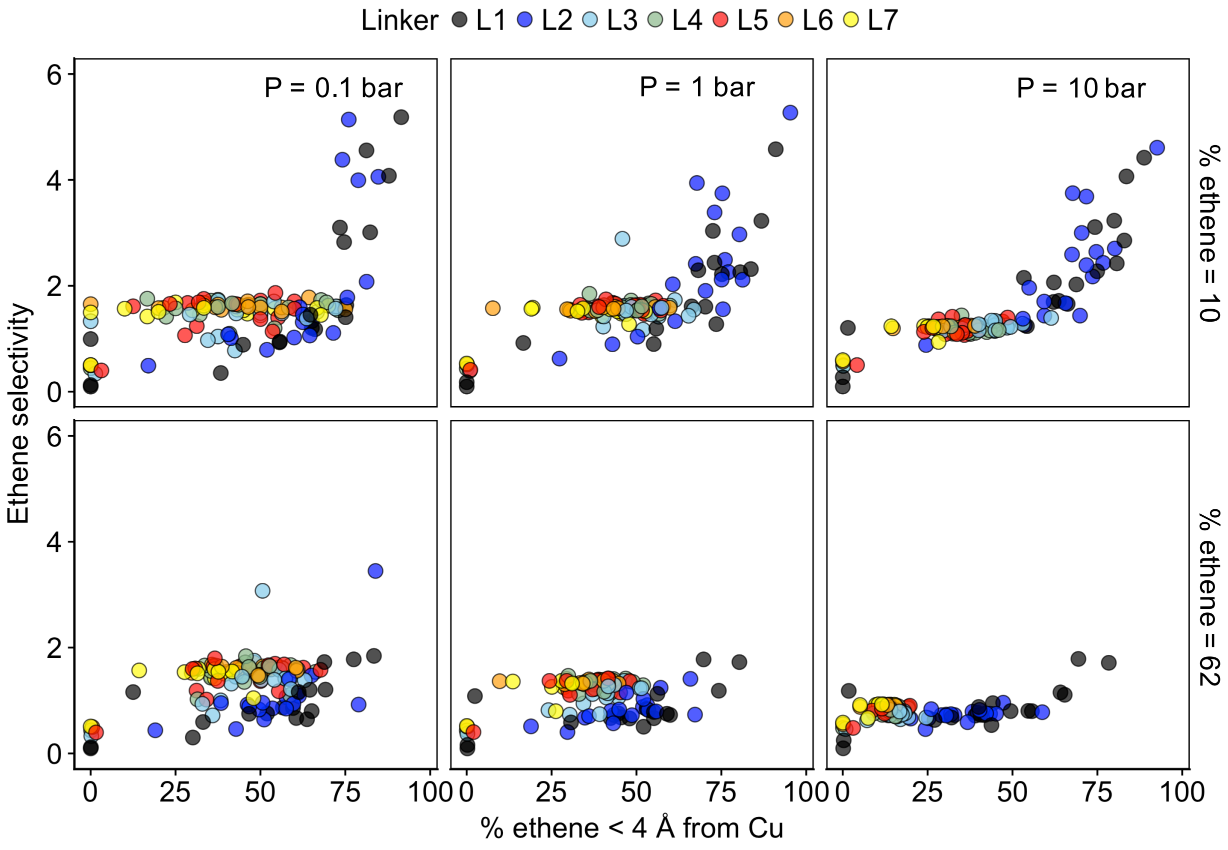

Figure S14. Plots of ethene selectivity versus the percentage of ethene (out of the total adsorbed ethene) less than 4 Å from a Cu OMS for each combination of pressure and mixture composition considered.

References

1 J. G. Eon, *Acta Crystallogr. Sect. A Found. Crystallogr.*, 2011, **67**, 68–86.
2 P. G. Boyd and T. K. Woo, *CrystEngComm*, 2016, **18**, 3777–3792.
3 O. Delgado-Friedrichs and M. O'Keeffe, *J. Solid State Chem.*, 2005, **178**, 2480–2485.
4 T. Kavitha, C. Liebchen, K. Mehlhorn, D. Michail, R. Rizzi, T. Ueckerdt and K. A. Zweig, *Comput. Sci. Rev.*, 2009, **3**, 199–243.
5 C. Zhu, R. H. Byrd, P. Lu and J. Nocedal, *ACM Trans. Math. Softw.*, 1997, **23**, 550–560.
6 R. H. Byrd, P. Lu, J. Nocedal and C. Zhu, *SIAM J. Sci. Comput.*, 1995, **16**, 1190–1208.
7 P. J. A. Cock, T. Antao, J. T. Chang, B. A. Chapman, C. J. Cox, A. Dalke, I. Friedberg, T. Hamelryck, F. Kauff, B. Wilczynski and M. J. L. de Hoon, *Bioinformatics*, 2009, **25**, 1422–1423.
8 Python Language Reference, version 2.7. Python Software Foundation.
9 X. Lin, I. Telepeni, A. J. Blake, A. Dailly, C. M. Brown, J. M. Simmons, M. Zoppi, G. S. Walker, K. M. Thomas, T. J. Mays, P. Hubberstey, N. R. Champness and M. Schröder, *J. Am. Chem. Soc.*, 2009, **131**, 2159–2171.
10 B. Chen, N. W. Ockwig, A. R. Millward, D. S. Contreras and O. M. Yaghi, *Angew. Chemie - Int. Ed.*, 2005, **44**, 4745–4749.
11 R. Hill, *Proc. Phys. Soc. Sect. A*, 1952, **65**, 349–354.