DESIA: A General Framework for Designing Interlocking Assemblies

ZIQI WANG, EPFL
PENG SONG, EPFL
MARK PAULY, EPFL

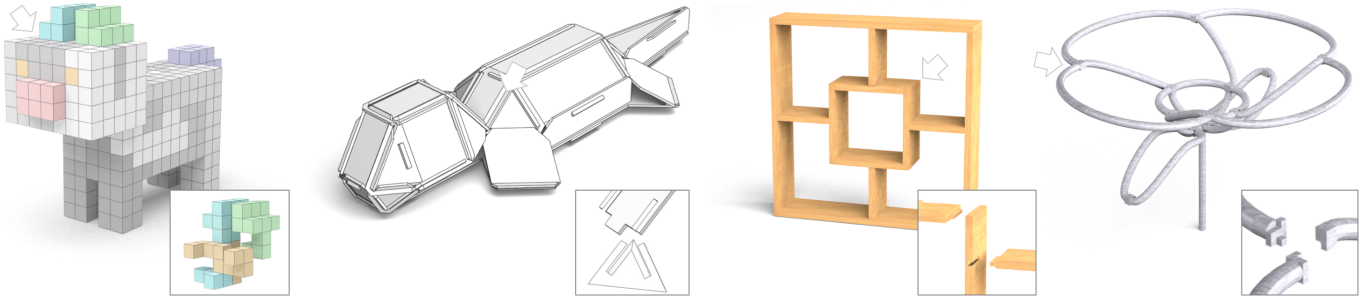


Fig. 1. Various interlocking assemblies designed using our framework, from left to right: voxelized puzzle, plate structure, furniture, and frame structure. Our method supports different types of joints as highlighted in the zooms. Please refer to the accompanying video for assembly sequences and the supplementary material for the blocking graphs defining the interlocking configurations.

Interlocking assemblies have a long history in the design of puzzles, furniture, architecture, and other complex geometric structures. The key defining property of interlocking assemblies is that all component parts are immobilized by their geometric arrangement, preventing the assembly from falling apart. Computer graphics research has recently contributed design tools that allow creating new interlocking assemblies. However, these tools focus on specific kinds of assemblies and explore only a limited space of interlocking configurations, which restricts their applicability for design.

In this paper, we propose a new general framework for designing interlocking assemblies. The core idea is to represent part relationships with a family of base *Directional Blocking Graphs* and leverage efficient graph analysis tools to compute an interlocking arrangement of parts. This avoids the exponential complexity of brute-force search. Our algorithm iteratively constructs the geometry of assembly components, taking advantage of all existing blocking relations for constructing successive parts. As a result, our approach supports a wider range of assembly forms compared to previous methods and provides significantly more design flexibility. We show that our framework facilitates efficient design of complex interlocking assemblies, including new solutions that cannot be achieved by state of the art approaches.

CCS Concepts: • **Computing methodologies** → *Shape modeling*; • **Applied computing** → *Computer-aided manufacturing*;

Additional Key Words and Phrases: 3D assembly, interlocking, component parts, joints, computational design, directed graph

0730-0301/2018/11-ART191 $15.00
https://doi.org/10.1145/3272127.3275034

ACM Reference format:
Ziqi Wang, Peng Song, and Mark Pauly. 2018. DESIA: A General Framework for Designing Interlocking Assemblies. *ACM Trans. Graph.* 37, 6, Article 191 (November 2018), 14 pages.
https://doi.org/10.1145/3272127.3275034

1 INTRODUCTION

3D assemblies refer to objects that combine multiple component parts into a structure with a specific form and/or functionality. Connection mechanisms are usually required to prevent the parts from moving relative to one another and make the assembly steady for practical use. However, these connectors can be irreversible (e.g., glue), impair the structural integrity of parts (e.g., nails), or degrade the external appearance of the assembly (e.g., clamps).

Rather than relying on additional explicit connectors, interlocking assemblies connect parts into a steady structure based only on the geometric arrangement of the parts. This intriguing property facilitates repeated assembly and disassembly and significantly simplifies the correct alignment of parts during construction. Consequently, interlocking assemblies have been used in a variety of applications, including puzzles [Stegmann 2018], furniture [Fu et al. 2015], architecture [Deepak 2012], and 3D printing [Yao et al. 2017a].

In an interlocking assembly, parts need to follow certain orders to be assembled into the target object. Once assembled, there is only one movable part, called the *key*, while all other parts as well as any subset of parts are immobilized relative to one another [Song et al. 2012]. However, this defining property of parts immobilization makes designing interlocking assemblies highly challenging. Explicitly testing the immobilization of every subset of parts requires costly computations; optimizing for the geometry of parts that satisfy these immobilization requirements, while avoiding deadlocking, is even more complex.

Recently, several computational approaches have been developed to address this problem [Fu et al. 2015; Song et al. 2016, 2012, 2017; Xin et al. 2011; Yao et al. 2017a; Zhang and Balkcom 2016]. The common idea is to directly guarantee global interlocking by constructing and connecting multiple local interlocking groups (LIGs), which avoids the overhead of testing all part subsets for immobilization. While these methods show successful results, they only focus on specific sub-classes of interlocking assemblies, e.g., recursive interlocking puzzles [Song et al. 2012], but do not explore the full search space of all possible interlocking configurations. As a consequence, these approaches are restricted in the kind of input shapes they can handle and have limited flexibility to satisfy additional design requirements besides interlocking, e.g., related to aesthetics or functional performance.

Contributions. In this paper, we propose a new general framework for DESigning Interlocking Assemblies, called *DESIA*, that avoids the restrictions of previous LIG-based methods. Specifically, we make the following contributions:

- We represent interlocking assemblies with a set of base *Directional Blocking Graphs* (DBGs) and implement an efficient graph analysis algorithm that can test for global interlocking in polynomial time complexity.
- We introduce a general iterative framework for designing interlocking assemblies that can explore the full search space of all possible interlocking configurations by utilizing all existing part blocking relations described in the graphs.
- We demonstrate the flexibility of our framework for designing different classes of assemblies, including new types of interlocking forms that have not been explored in previous works.

The rest of the paper is organized as follows. We first discuss related work in Section 2. In Section 3 we introduce our graph-based representation for assemblies and present efficient algorithms for testing whether an assembly is interlocking. Section 4 then describes our computational framework for designing interlocking assemblies. In Section 5 we show different types of assemblies generated with our approach, compare with previous works, and highlight several application examples. We conclude with a discussion of limitations of our approach and some thoughts on future research problems.

2 RELATED WORK

Connecting Parts in 3D Assemblies. To create a multi-part object for practical use, component parts need to be assembled and tightly connected. For example, glue is used to connect 3D printed parts [Chen et al. 2015; Hu et al. 2014; Vanek et al. 2014] although glued parts cannot be separated easily, discouraging parts disassembly and reassembly. Nails and screws are commonly used in furniture [Lau et al. 2011; Shao et al. 2016]. However, these fasteners may break the parts and become loose after repeated disassembly and reassembly. Wire also can be used for part connections [Attene 2015; Richter and Alexa 2015], yet tying parts together could be a tedious task.

In practice, integral joints are often preferred for connecting parts, since they greatly simplify the assembly process. For example, woodworking joints (see Figure 2) are widely used in furniture

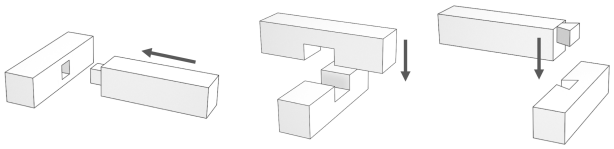


Fig. 2. Example woodworking joints. From left to right: mortise-and-tenon, halved joint, and dovetail joint, where the black arrow shows the single movable direction of the part allowed by the joint.

design [Chen and Sass 2015; Koo et al. 2017; Schwartzburg and Pauly 2013], mortise-and-tenon joints for 3D printed object assemblies [Duncan et al. 2016; Hao et al. 2011; Luo et al. 2012], and halved joints for laser-cut shape abstractions [Cignoni et al. 2014; Hildebrand et al. 2012; McCrae et al. 2014]. However, these joints only constrain relative part motion locally, and the resulting assembly typically relies on other means, e.g., friction and/or gravity, to lock the parts in place [Yao et al. 2017b].

Self-supporting Structures, e.g., masonry buildings [Rippmann et al. 2016] and puzzles [Frick et al. 2015], are assemblies of rigid components that do not require any binder to connect the parts. The entire structure is in static equilibrium through gravity-induced compression forces that immobilize all the parts [Whiting et al. 2009]. Recently, the design of freeform self-supporting structures has received a lot of interest in computer graphics. Some research works focus on designing 3D surfaces that only exhibit internal compression forces under gravity [de Goes et al. 2013; Liu et al. 2013; Miki et al. 2015; Tang et al. 2014; Vouga et al. 2012], while others focus on the design, fabrication, and assembly of self-supporting structures [Deuss et al. 2014; Panozzo et al. 2013; Rippmann et al. 2016]. Self-supporting structures can only bear compression load but are not designed to handle any tensile force. This is reasonable for certain architectural structures that are built to bear their own weight, but not for general spatial assemblies that experience forces in arbitrary directions. The strategy of immobilizing parts in self-supporting structures is therefore not applicable in our general assembly setting.

Interlocking Assemblies. Several computational methods have recently been developed to construct interlocking assemblies. Xin et al. [2011] create 3D interlocking puzzles by replicating and connecting multiple instances of a six-piece interlocking burr structure. Rather than reusing an existing structure, Song et al. [2012] construct 3D interlocking puzzles by iteratively extracting pieces from a voxelized 3D shape and enforcing a local interlocking condition among every three consecutive pieces. Song et al. [2015] extend this method to handle smooth non-voxelized shapes for 3D printing. Zhang and Balkcom [2016] define a set of voxel-like interlocking blocks and connect instances of these blocks layer-by-layer into various voxelized shapes.

Different from above works that take a 3D solid object as an input, Fu et al. [2015] focus on plate structures such as furniture that have been initially partitioned into parts. They compute an interlocking joint configuration by planning and connecting local interlocking groups. This method has been extended to interlock 2D laser-cut parts into a convex polyhedron [Song et al. 2016] and to design reconfigurable furniture with multi-key interlocking [Song et al.

2017]. Yao et al. [2017a] design and optimize interlocking shell pieces for 3D printing. They employ 3D simulation to test whether the resulting shell assemblies are interlocking. However, this generate-and-test paradigm is relatively inefficient and applicable only for assemblies with a small number of interlocking pieces.

The key idea of the above works (except [Yao et al. 2017a]) is to design interlocking assemblies by constructing and connecting multiple local interlocking groups. This strategy skillfully avoids the test for global interlocking of the resulting assemblies, which would have a time complexity that is exponential in the number of parts [Song et al. 2012]. While this strategy allows designing interlocking assemblies with many parts, the search space is restricted to a small subset of all possible interlocking configurations. Our experimental results in Section 5 show that the flexibility of designing interlocking assemblies can be significantly increased by our method's ability to explore the full search space.

Assembly Planning is the problem of finding a sequence of motions to assemble a structure from its parts. This problem has been extensively studied in robotics and we refer to [Ghandi and Masehian 2015] for a thorough review. Assembly planning is also relevant for computer graphics applications, e.g., for generating visual assembly instructions [Agrawala et al. 2003; Guo et al. 2013] or creating exploded view diagrams [Li et al. 2008].

Finding an assembly plan requires identifying movable parts and part groups at each intermediate assembly state, often leading to a combinatorial search problem. To solve this task more efficiently, Wilson [1992] invented a *Directional Blocking Graph* and a *Non-Directional Blocking Graph* to represent blocking relations among parts in an assembly. Tai [2012] employed these graphs for designing reciprocal frame structures connected with notched joints, aiming at minimizing the number of movable parts and part groups in the final assembly. Rather than focusing on assembly planning, our work, to the best of our knowledge, is the first to employ these graphs for computational design of interlocking assemblies. Specifically, we address the challenges of 1) efficiently testing for interlocking and 2) constructing the geometry of interlocking parts.

3 MODEL INTERLOCKING ASSEMBLIES

In this section, we introduce our conceptual representation of interlocking assemblies using a family of directed graphs. We show how this graph-based representation leads to efficient algorithms to *test* interlocking. In Section 4 we then explain how to effectively employ this representation and algorithms to *design* interlocking assemblies.

3.1 Graph Model

Consider an assembly **A**, made of N parts $P_1, ..., P_N$. We make the following assumptions: 1) each part P_i is rigid; 2) neighboring parts have planar surface contact only; and 3) **A** can be disassembled by single-part translational motions, i.e., part rotation is not required and all other parts remain fixed when removing a part.

Directional Blocking Graph (DBG). We denote as $G(d, A)$ the *directional blocking graph* of assembly **A** for translation along direction d. This directed graph has nodes representing the parts of **A** and directed edges $e_{i\to j}$ from P_i to P_j if and only if P_j prevents any

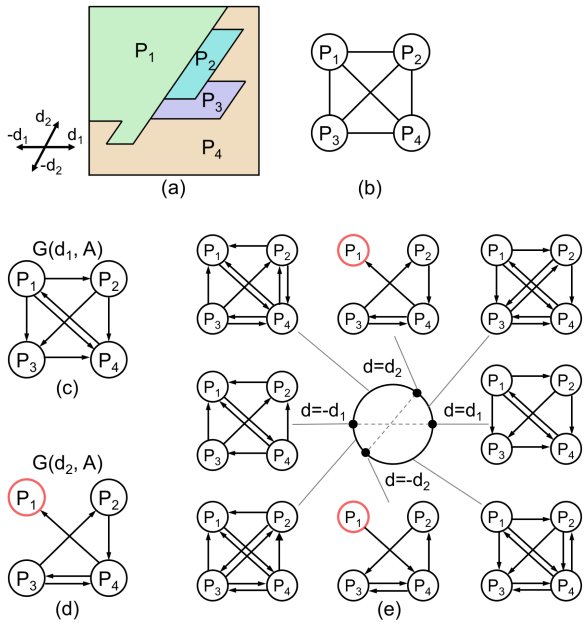
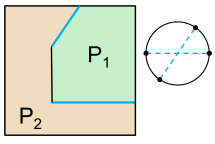


Fig. 3. Example DBGs and NDBG. (a&b) A 2D interlocking assembly and its parts-graph, where the key P_1 is movable along d_2; (c&d) Two DBGs of the assembly; and (e) NDBG of the assembly. A part with zero out-degree or in-degree in a DBG is highlighted with a red circle.

translational motion of P_i along d. In other words, $e_{i\to j}$ can be read as "P_i is blocked by P_j" in direction d. See Figure 3(c&d) for two examples.

If $G(d, A)$ is *strongly connected*, i.e. if every node can be reached from every other node, no part or part group is movable along d; see Figure 3(c). A part group **S** of **A** is locally free to translate in direction d ($-d$), if and only if the out-degree (in-degree) of **S** in $G(d, A)$ is zero; see P_1 in Figure 3(d).

Non-directional Blocking Graph (NDBG). We represent the set of all translation directions in 2D by the unit circle denoted as C. For every pair of parts in contact in **A**, we draw the diameter that is parallel with the contact line. The drawn diameters partition C into an arrangement of regions, for which the corresponding DBG $G(d, A)$ remains constant when d varies over a region. For any pair of parts in contact (e.g., P_1 and P_2 in the inset), if there are more than two contact lines, we only retain the two diameters of C (e.g., two contact lines in blue) which bound the cone of directions in which one part is free to translate relative to the other. The arrangement of points and intervals on C, and the associated DBGs form the *non-directional blocking graph* of **A**; see Figure 3(e). An NDBG of a 3D assembly can be built similarly by constructing DBGs for each point and regular region on a unit sphere that represents all possible translation directions in 3D; please refer to [Wilson and Latombe 1994] for more details.



Base Directional Blocking Graphs. An NDBG represents the parts blocking relations with redundancy in two aspects. First, the DBG corresponding to an arc in C can be derived by performing union operations on the DBGs associated with the two end points of the

arc; see again Figure 3(e). Second, we can obtain $G(-d, A)$ from $G(d, A)$ easily by reversing the direction of every edge in $G(d, A)$ due to the reciprocity of blocking relations among the parts.

Therefore, it is sufficient to model the blocking relations in $\mathbf{A}$ by using only a set of *base DBGs* denoted as $\{G(d, A)\}$, which we select as the DBGs corresponding to the end points in a half circle of C. For example, two DBGs in Figure 3(c&d) form $\{G(d, A)\}$. We call the set of directions corresponding to the base DBGs as *base directions*, denoted as $\{d\}$. The number of base DBGs (as well as base directions) is $O(N^2)$ since every pair of parts provides at most two diameters in C.

3.2 Testing Interlocking

In an interlocking assembly, every part and every part group are immobilized for all possible translation directions, except a single key. A part group is not immobilized if parts in the group are able to 1) translate along the same direction; or 2) different directions simultaneously (see Figure 4). In this subsection, we develop two approaches for testing interlocking of 3D assemblies: a DBG-based approach that considers only the first kind of part group movement; and an inequality-based approach that considers both kinds of part group movement.

DBG-based Testing Approach. To test immobilization of a part group $\mathbf{S}$, we need to compute blocking relations between $\mathbf{S}$ and $\mathbf{A} - \mathbf{S}$: the part group $\mathbf{S}$ is immobilized if $\mathbf{S}$ is blocked by $\mathbf{A} - \mathbf{S}$ in all translation directions. Explicitly testing interlocking by checking immobilization of every part and every part group has exponential time complexity. However, treating each part group $\mathbf{S}$ independently ignores significant redundancies in the blocking relations across the parts. We exploit these redundancies and propose a more efficient approach to test interlocking. The key idea is to utilize the blocking relations encoded in the set of base DBGs to implicitly test immobilization of every part and every part group along a finite number of translation directions, i.e., the base directions $\{d\}$.

In detail, an assembly with at least three parts is interlocking, if all base DGBs are either

(1) strongly connected, or
(2) have only two strongly connected components one of which has a single part that is identical across all DGBs.

Here the strongly connected component with a single part is the key of the assembly. Direction d associated with each DBG with two strongly connected components is the key's (reversed) movable direction according to the in-edge (out-edge) of the key in the DBG; e.g., the assembly in Figure 3(a) is interlocking since its two base DBGs in Figure 3(c&d) satisfy the above requirement.

In our implementation, we use Tarjan's algorithm [Tarjan 1972] to find strongly connected components in each DBG. Runtime complexity is linear in the number of edges and nodes in the graph, i.e., $O(N^2)$ since there are at most N^2 edges in the graph. As the set of base DBGs has $O(N^2)$ graphs, the worst-case complexity of our interlocking testing algorithm is $O(N^4)$, which is much lower than $O(2^N)$ of the previous approach [Song et al. 2012]. In particular, the complexity to test interlocking of a well-structured assembly, where each part connects with at most $L \ll N$ parts, is $O(L^2N^2)$ since the number of base DBGs is $O(LN)$ and running Tarjan's algorithm on each DBG is also $O(LN)$. In practice, our algorithm is extremely fast. For example, our implementation can test for interlocking of the 80-part Bunny assembly in Figure 12(b) in 0.5 milliseconds, including the construction of all blocking graphs. For comparison, the approach of [Song et al. 2012] takes 24.6 seconds on a 20-part Bunny of the same voxel count and would run years on the 80-part model; see also Section 5.

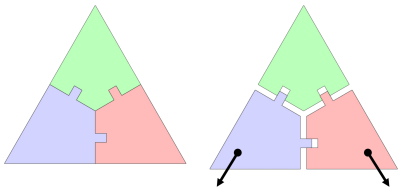


Fig. 4. A 2D assembly for which translating two parts along different directions (black arrows) simultaneously is the only way to disassemble it.

This DBG-based approach is sufficient for testing interlocking of 3D assemblies where parts are orthogonally connected; see supplementary material for a proof. However, it is only necessary but not sufficient for testing interlocking of 3D assemblies with non-orthogonal part connections. Figure 4 shows a counter example. The DBG-based approach identifies this assembly as deadlocking yet two parts actually can move along different directions simultaneously. To address this issue, we devise the following inequality-based approach.

Inequality-based Testing Approach. Consider that each part P_i can translate freely in 3D space and denote the velocity of P_i as v_i. During the parts movement, the constraint is to avoid collision among the parts. We model this collision free constraint between P_i and P_j as $(v_j - v_i) \cdot n_{ij} \geq 0$, where n_{ij} is the normal of the planar contact interface between P_i and P_j (pointing towards P_j).

By stacking all these constraints, we get a system of linear inequalities

$$AV \geq 0, \tag{1}$$

where $V = [v_1, ..., v_n]^T$, and A is the matrix specifying the coefficients given by the normals of all interfaces. To avoid the case that all parts move together as a whole, we randomly select a part, say P_r, as a reference, and fix it by setting its velocity $v_r = 0$.

We consider the assembly as deadlocking if the system does not have any non-zero solution. However, directly solving the system is nontrivial due to the high dimensional search space (i.e., number of variables larger than 3) [Solodovnikov 1979]. Instead, we address the problem by formulating a linear program with some auxiliary variables.

$$\max(\sum t_{ij}) \tag{2}$$

$$\begin{aligned} \text{s.t.} \quad & (v_j - v_i) \cdot n_{ij} \geq t_{ij}, \quad \forall \text{interfaces}(i, j) \\ & 0 \leq t_{ij} \leq 1, \\ & v_r = 0. \end{aligned}$$

If the assembly is deadlocking, then all t_{ij} should be 0; otherwise, some t_{ij} should be strictly larger than 0.

A 3D assembly is interlocking if it satisfies: 1) there is only one movable part, say P_k; and 2) the assembly is deadlocking if we fix P_k. Hence, we first run the above linear program while setting velocities of all the parts as zero except P_i. We iterate this simplified

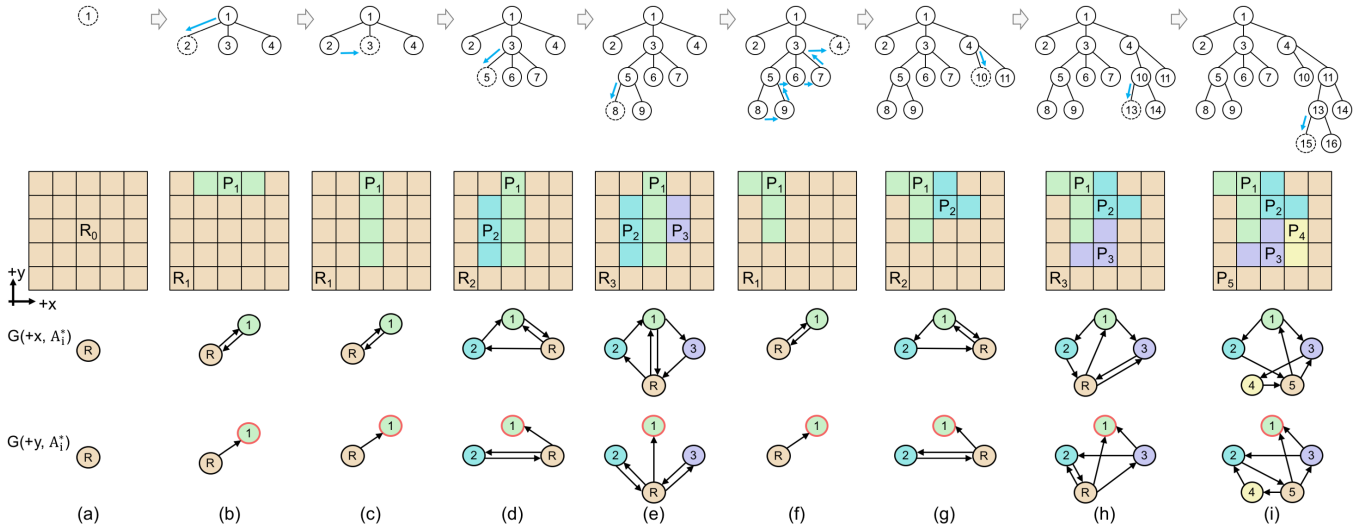


Fig. 5. Overview of our framework on designing a 2D interlocking assembly. (a) Given a 5×5 square as input, (b-i) our framework tries to generate a 5-part interlocking 2D assembly, in which each part should have at least two pixels. Top row: construction tree, where each node at depth i represents a candidate of $\mathbf{A}_i$. Here we show the $m = 3$ highest ranked options at each level with the top-ranked on the left. The blue arrows indicate the procedure to visit the nodes for generating parts. If the framework cannot find any child for the current node (in dashed circle, denoted as $\mathbf{A}_i^*$), it will backtrack to (c) its siblings or (f) ancestors. Middle row: geometric examples corresponding to the dashed node in the tree. Bottom row: base DBGs of the geometric examples, where the key is indicated by a red circle. For simplicity, we show nodes P_j ($j \leq i$) as j, and R_i as R in (a-h), and show the last part R_4 as P_5 in the final assembly (i).

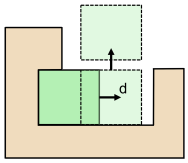
linear program for each P_i, and identify all the parts that can move individually. We consider the assembly as non-interlocking if the number of such parts is not one. Next, we run the above linear program while setting velocities of the single movable part P_k and the reference part P_r ($P_r \neq P_k$) as zero. We consider the assembly as interlocking if the linear program cannot find any non-zero solution.

We have implemented both testing approaches and found that the DBG-based approach is much faster than the inequality-based approach; e.g., it takes 0.0076 and 0.1281 seconds respectively on a 1500-part Cube assembly (see Figure 12(d)). Thus, for assemblies with non-orthogonal part connections, it would be more efficient to run the DBG-based test first to filter out non-interlocking assemblies, followed by verification with the inequality-based test.

3.3 Testing Disassemblability

The DBGs were originally developed to test whether a structure can be (dis)assembled. By successively identifying the movable part (or part group) based on the DBGs, we can find a possible sequence to disassemble all the parts. Otherwise, we consider the structure as not (dis)assemblable by part translational motions.

In practice, a part P may not be able to be taken out by a single translation, say along d, since some part may block P after it moves along d for a certain distance; see the inset figure. Since this case is not modeled in the DBGs, we try to find a collision-free path for taking out P by sampling P's position along the translation direction and allowing changing P's translation direction at a certain point. Otherwise, we consider P as not disassemblable. Alternatively, more complex disassembly path planning approaches, e.g. [Ghandi and Masehian 2015], could be employed here.



4 COMPUTATIONAL DESIGN FRAMEWORK

Given our efficient algorithms to test for interlocking, our main goal is now to provide effective algorithms for designing interlocking assemblies. We first provide a high-level overview of our framework before presenting the conceptual and algorithmic details.

As input we expect the final shape of the assembly, from which the component parts are either constructed from scratch as in [Song et al. 2012, 2015; Xin et al. 2011] or explicitly initialized as in [Fu et al. 2015; Song et al. 2016; Yao et al. 2017a]. Our computational process for creating an interlocking assembly starts with the full input model, then iteratively splits off successive parts for disassembly. At each iteration, we first identify a set of suitable blocking relations to be generated between the current assembly and the new part such that the interlocking property is maintained. Then we search for the part geometry that satisfies these blocking relations. The selection of a new part is guided by a ranking function that takes into account certain geometric properties, e.g. part size, or other requirements, e.g. on part fabrication. The search space is then explored in a tree traversal process that uses automatic backtracking when no interlocking solution could be found in a specific iteration; see Figure 5. We also provide a user interface to interactively explore different options for part decomposition, allowing the user to overwrite the generic ranking function for part selection; see Figure 15.

4.1 Iterative Design Framework

Given the input shape denoted as R_0, we iteratively construct the geometry of each part (or introduce appropriate joints in the geometry of each initialized part; see Section 5.2), one by one. This forms a sequence of constructed parts, $P_1, P_2, \ldots, P_n$, with R_n, the remaining part of R_0, as the last part:

$$[R_0] \rightarrow [P_1, R_1] \rightarrow [P_1, P_2, R_2] \rightarrow \ldots \rightarrow [P_1, \ldots P_n, R_n] .$$

Algorithm 1 Algorithm to design an interlocking assembly $\mathbf{A}_n$ from a given shape R_0.

```
1:  function CreateInterlockAssembly(R_0)
2:      i ← 0
3:      A_i^* ← [R_0]
4:      while i < n do
5:          if i=0 then
6:              {A_{i+1}} ← GenerateKey(A_i^*)      ▷ See Subsection 4.2
7:              if {A_{i+1}} = ∅ then
8:                  return NULL
9:          else
10:             {A_{i+1}} ← GenerateParts(A_i^*)    ▷ See Subsection 4.3
11:         if {A_{i+1}} ≠ ∅ then
12:             RankCandidates({A_{i+1}})           ▷ In descending order
13:             if i + 1 = n then
14:                 return A_n^1
15:             else
16:                 i ← i + 1
17:                 A_i^* ← A_i^1
18:         else if A_i^*.sibling ≠ NULL then
19:             A_i^* ← A_i^*.sibling   ▷ A_i^*.sibling is the one second to
    A_i^* in the ranked {A_i}
20:         else
21:             while A_i^*.parent ≠ NULL &&
22:                   A_i^*.parent.sibling = NULL do
23:                 A_i^* ← A_i^*.parent
24:                 i ← i − 1
25:                 if i = 0 then               ▷ Quit if backtrack to A_0
26:                     return NULL
27:             if A_i^*.parent ≠ NULL &&
28:                A_i^*.parent.sibling ≠ NULL then
29:                 A_i^* ← A^*.parent.sibling
30:             else
31:                 return NULL
```

Here we denote each intermediate assembly $[P_1, ..., P_i, R_i]$ as $\mathbf{A}_i$ ($0 \le i \le n$), and its base DBGs as $\{G(d, A_i)\}$. Figure 5 shows an example where the parts are constructed from scratch.

To guarantee that the resulting assembly $\mathbf{A}_n = [P_1, ..., P_n, R_n]$ is interlocking and disassemblable, we have the following requirements when decomposing R_{i-1} into P_i and R_i:

(i) *Connected.* The geometries of P_i and R_i should each be connected, making $\mathbf{A}_i$ a valid assembly.

(ii) *Interlocking.* $\mathbf{A}_i$ ($i \ge 2$) is interlocking with P_1 as the key. In other words, $\{G(d, A_i)\}$ should satisfy the interlocking requirement described in Section 3.

(iii) *Disassemblable.* P_i can be removed from $[P_i, R_i]$, so we can disassemble $\mathbf{A}_i$ in the order of P_1, P_2, ..., P_i, R_i.

The advantage of this iterative design framework is that we achieve the goal of global interlocking by satisfying a set of local requirements when constructing each pair of P_i and R_i.

Tree Traversal. Since we cannot guarantee that the construction of P_i and R_i succeeds at every iteration, we propose an iterative approach with backtracking to construct $\mathbf{A}_n$; see Figure 5 and Algorithm 1. The key idea is to build and maintain a construction tree, where each node represents a candidate of $\mathbf{A}_i$. For each node, we generate a set of children denoted as $\{\mathbf{A}_{i+1}\}$, among which the only different parts are P_{i+1} and R_{i+1}. Our approach ranks these candidate assemblies at each iteration to facilitate the construction of successive parts. For example, we rank $\{\mathbf{A}_{i+1}\}$ according to the compactness of R_{i+1} measured by using the accessibility in [Song et al. 2012], since parts extracted from a compact R_{i+1} are more likely to be connected; compare R_1 in Figure 5(c&f). In case the user has other design goals besides interlocking, e.g., regarding the appearance of the assembly, we support user intervention to adjust the ranking; see again Figure 15. In case we cannot generate any valid result from the selected candidate in $\{\mathbf{A}_{i+1}\}$, we can backtrack the tree to try other nodes without restarting the whole design process. The size of $\{\mathbf{A}_{i+1}\}$ is denoted as m. A large m requires more time for generating $\{\mathbf{A}_{i+1}\}$, but also provides more choices for ranking and backtracking. We set $m = 30$ by default in our experiments but it can be adjusted, depending on the input model.

Below we explain our approach to generate the key part (Subsection 4.2) and the remaining parts of the assembly (Subsection 4.3). These steps can be customized to design different kinds of interlocking assemblies as discussed in Section 5. Here, we take 2D interlocking puzzle design as an example for illustration.

4.2 Generating the key

We first partition the input model R_0 into P_1 and R_1, where P_1 is the key and R_1 is the remaining part. We construct the geometry of P_1 following the procedure in [Song et al. 2012], i.e., select a seed pixel, ensure its blocking and mobility, and expand the key part. Recall that the key is the only movable (thus unstable) part in an interlocking assembly. Therefore, we restrict P_1 to have a single movable direction in $\mathbf{A}_1$ denoted as d_1, and usually select d_1 being upward to stabilize P_1 with gravity; see Figure 5(b&c) for two examples. We rank the candidates in $\{\mathbf{A}_1\}$ according to the compactness of R_1.

4.3 Generating P_i and R_i ($i > 1$)

Next, we construct P_i and R_i from R_{i-1} in two stages: *graph design* and *geometry realization*. The first stage constructs base DBGs $\{G(d, A_i)\}$ that satisfy the interlocking requirement conceptually. And the second stage aims at realizing the blocking relations described in $\{G(d, A_i)\}$ in the embedded geometry while satisfying the part connectivity and disassemblability requirements defined in Subsection 4.1. Note that geometric constraints (e.g., supported joint types) can be used to simplify graph design by eliminating potential graph edges that cannot be realized geometrically anyway.

Graph Design for P_i and R_i. Starting from $\mathbf{A}_{i-1}$ ($i \ge 2$), the goal is to find blocking relations for P_i and R_i such that the updated assembly $\mathbf{A}_i$ is still interlocking. In other words, after splitting R_{i-1} into P_i and R_i in $\{G(d, A_{i-1})\}$ to form $\{G(d, A_i)\}$, we need to construct a set of new edges for P_i and R_i in each $G(d, A_i)$ such that the graph remains strongly connected, except the key; see Figure 5. To achieve this goal, we first classify blocking relations to be constructed into two classes:

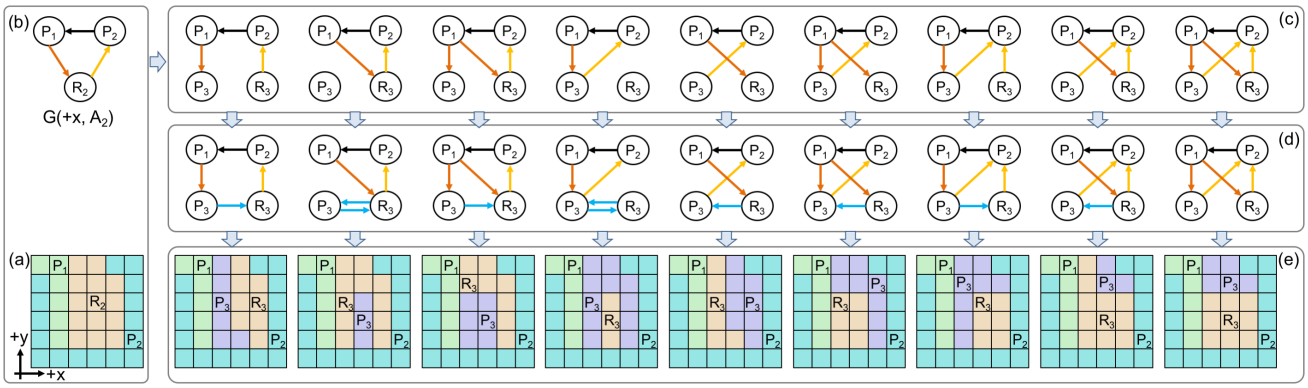


Fig. 6. (a) An intermediate assembly $\mathbf{A}_2$ and (b) its base DBG $G(+x, A_2)$, where the in-edge and out-edge of R_2 are colored in dark and light orange respectively; (c) all cases of distributing existing blocking relations (dark and light orange edges) to P_3 and R_3; (d) the interlocking graph designs that require the fewest internal blocking relations (blue edges) between P_3 and R_3; and (e) the corresponding geometric examples.

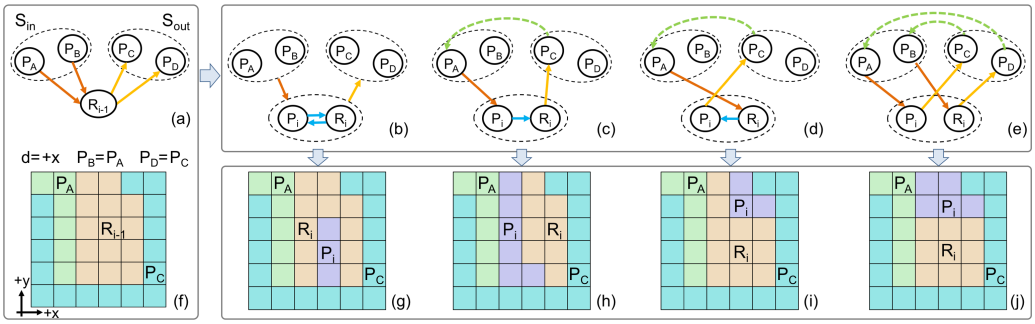


Fig. 7. (a) Given $G(d, A_{i-1})$, (b-e) we ensure that $G(d, A_i)$ is strongly connected by constructing a cycle that includes both P_i and R_i. A dashed ellipse in (a-e) indicates a subset of parts, i.e., S_{in}, S_{out}, and $\{P_i, R_i\}$, where S_{in} (S_{out}) denotes the set of parts with an edge to (from) R_{i-1}. The directed edges from (to) a dashed circle in (b) indicate that the edge can be from (to) any part in the associated subset. The dashed green edges in (c-e) indicate that a part can reach the other part in $G(d, A_{i-1})$ without passing through R_{i-1}. (f-i) Geometric examples corresponding to (a-e), where $d = +x$, $S_{in} = \{P_A\}$, and $S_{out} = \{P_C\}$.

(i) *External blocking relations* between $\{P_1, .., P_{i-1}\}$ and P_i, as well as those between $\{P_1, .., P_{i-1}\}$ and R_i are inherited from those between $\{P_1, .., P_{i-1}\}$ and R_{i-1}. We need to distribute these existing blocking relations to P_i and R_i; see Figure 6(c).

(ii) *Internal blocking relations* between P_i and R_i. For each case of distributing external blocking relations, we may need to construct internal blocking relations between P_i and R_i such that each $G(d, A_i)$ remains strongly connected[1]; see Figure 6(d).

Given these observations, we could find all valid graph designs by enumerating the distribution of external blocking relations, constructing the corresponding internal blocking relations, and testing the strongly connected property of the DBGs. However, this generate-and-test approach could be very inefficient. The number of choices to distribute external blocking relations is 3^l, where l is the number of edges of R_{i-1} in $G(d, A_{i-1})$, since each edge of R_{i-1} can be distributed to P_i, R_i, or both. Figure 6 shows an example with $3^2 = 9$ graph designs, where $l = 2$.

Rather than enumerating all possible graph designs, we propose an efficient approach to find a desired number of designs that are interlocking conceptually; see Figure 7. The key idea is to directly guarantee that each $G(d, A_i)$ is strongly connected by constructing a cycle in the graph that includes both P_i and R_i, given that $G(d, A_{i-1})$ is already strongly connected. Denote S_{in} (S_{out}) as the set of parts with an edge to (from) R_{i-1}, and P_{in} (P_{out}) as an arbitrary part in S_{in} (S_{out}); see Figure 7(a&f). According to the number of internal blocking relations to be constructed between P_i and R_i denoted as K, we have the following three cases to construct the cycle that we can choose independently for each DBG.

(1) *K=2.* $P_i \rightarrow R_i \rightarrow P_i$ forms a cycle, i.e., any distribution of external blocking relations works for this case; see Figure 7(b).

(2) *K=1.* $P_i \rightarrow R_i \rightarrow P_{out} \dashrightarrow P_{in} \rightarrow P_i$ ($R_i \rightarrow P_i \rightarrow P_{out} \dashrightarrow P_{in} \rightarrow R_i$) forms a cycle if the single directed edge is from P_i to R_i (from R_i to P_i); see Figure 7(c&d). Here, $P_{out} \dashrightarrow P_{in}$ means that P_{out} can reach P_{in} in $G(d, A_{i-1})$ without passing through R_{i-1}, or P_{out} and P_{in} are the same part.

(3) *K=0.* $P_i \rightarrow P_{out} \dashrightarrow P_{in} \rightarrow R_i \rightarrow P'_{out} \dashrightarrow P'_{in} \rightarrow P_i$ forms a cycle, where P_{in} and P'_{in} (as well as P_{out} and P'_{out}) are possible to be the same part; see Figure 7(e).

Compared with case 1, cases 2 and 3 rely more on external blocking relations than on internal blocking relations to immobilize P_i and R_i. As a consequence, these two cases impose fewer constraints on the subsequent geometry construction of P_i and R_i, resulting in a

[1]If the key is movable along d, $G(d, A_i)$ is strongly connected without considering the key. Otherwise, the whole graph of $G(d, A_i)$ should be strongly connected.

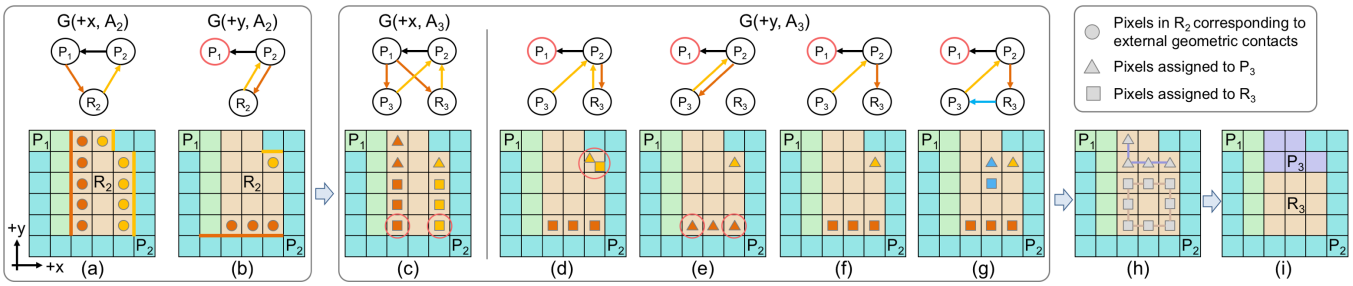


Fig. 8. Geometry realization of $\{G(d, A_3)\}$. (a&b) Identify geometric contacts between $\{P_1, P_2\}$ and R_2 in $\mathbf{A}_2$. (c-f) Distribute external geometric contacts between P_3 (shows as triangles) and R_3 (shown as squares) for (c) $G(+x, A_3)$ and (d-f) $G(+y, A_3)$, where (d&e) show two failure examples. (g) Realize internal blocking relation between P_3 and R_3 in $G(+y, A_3)$. (h) Construct initial geometry of P_3 and R_3. (i) Resulting $\mathbf{A}_3$.

higher chance to be successfully realized in the embedded geometry; compare the geometric examples in Figure 7(g-j).

Besides interlocking, we also need to ensure that P_i is disassemblable in $[P_i, R_i]$. Thus, we require that there are fewer than two directed edges between P_i and R_i (i.e., case 2 and 3) in at least one base DBG. The output of this stage is a set of $\{G(d, A_i)\}$ that satisfy the interlocking requirement, denoted as $\mathbf{C}_i$.

Geometry Realization of P_i and R_i. In order to realize $\{G(d, A_i)\} \in \mathbf{C}_i$ in the embedded geometry, we perform the following steps, each corresponding to a counterpart of the graph design stage:

i) Identify external geometric contacts between $\{P_1, ..., P_{i-1}\}$ and R_{i-1}. Recall that a directed edge $e_{i\to j}$ from P_i to P_j in $G(d, A)$ means that P_j blocks the translation of P_i along d. This indicates that P_i contacts P_j along d, and P_j locates further than P_i along d; see again Figure 3. In an assembly $\mathbf{A}_{i-1}$, we identify such blocking contacts between P_l $(1 \le l \le i-1)$ and R_{i-1} for each base direction d by computing the overlap of the respective boundaries along d $(-d)$, see Figure 8(a&b).

ii) Distribute external geometric contacts. An external blocking relation, say between P_l and R_{i-1}, in a DBG $G(d, A_{i-1})$ can be distributed to P_i, R_i, or both. For the first two cases, the corresponding geometric contacts need to be all assigned to P_i or R_i respectively; see Figure 8(f). For the last case, the geometric contacts need to be partitioned into two subsets and assigned to P_i and R_i separately; see Figure 8(c).

However, this step could fail for two reasons. First, the external geometric contact could be too small to be partitioned. For example, R_2 contacts P_2 along $+y$ with a single pixel in Figure 8 (b). Yet, this single pixel (marked with a red circle in Figure 8(d)) needs to be assigned to both P_3 and R_3 according to the computed blocking relations, which is not feasible. Second, the assignment of geometric contacts may conflict with one another across multiple DBGs. For example, two pixels marked with red circles in Figure 8(c) need to be assigned to R_3 to realize $G(+x, A_3)$. However, these two pixels also need to be assigned to P_3 to realize $G(+y, A_3)$ in Figure 8(e), leading to a conflict.

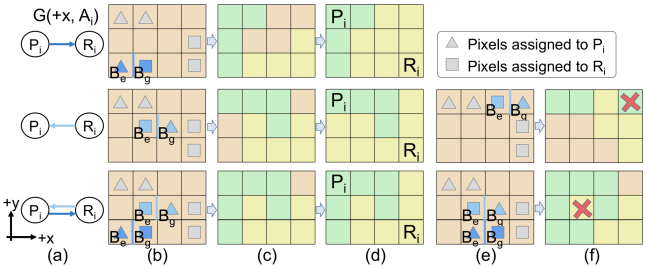


Fig. 9. (a) Internal blocking relations between P_i and R_i in $G(+x, A_i)$. (b) Find blocking and blockee pixels in R_{i-1} (in orange) according to the blocking relations, where blocking and blockee pixels in (b) and their associated blocking relation in (a) are colored the same (light or dark blue). (c) Initial geometry of P_i and R_i. (d) Final geometry of P_i and R_i. (e&f) Two failure examples due to disconnectivity of P_i or R_i (see the red cross).

iii) Construct internal geometric contacts. If $G(d, A_i)$ has $K \in \{1, 2\}$ internal blocking relations, we need to construct geometric contacts between P_i and R_i. Here, we take as an example the case of realizing a single directed edge from P_i to R_i to illustrate our approach; see Figure 9(top). Inspired by [Song et al. 2012], we find among all unassigned pixels in R_{i-1} a pair of blocking and blockee pixels that contact each other along d, denoted as B_g and B_e respectively. We then assign B_e to P_i and B_g to R_i. Other cases of realizing internal blocking relations can be handled similarly; see Figure 9.

iv) Construct initial parts geometry. By now, we have identified all the pixels in R_{i-1} that need to be assigned to P_i or R_i to make $\mathbf{A}_i$ interlocking. To form an initial P_i (R_i), we connect these pixels into a single part using the shortest path; see Figure 8(h) and 9(c). Note that this connection process can fail since we may not be able to find such a shortest path without disconnecting parts; see Figure 9(e&f) for examples.

v) Ensure disassemblability. To make P_i movable in $[P_i, R_i]$, we first identify all possible moving directions of P_i in $\{G(d, A_i)\}$, i.e., the directions where P_i is unblocked by R_i; e.g., P_3 could be movable along $\{-x, +x, +y\}$ in $[P_i, R_i]$ according to the blocking graph in Figure 8(c&g) . We try each possible moving direction of the initial P_i and discard those that cannot be achieved in the embedded geometry. We consider that P_i is disassemblable in $[P_i, R_i]$ if we can find one movable direction of P_i, along with a disassembly path. Lastly, we assign those remaining pixels in R_{i-1} (see orange pixels in Figure 9(c)) to P_i and R_i respectively according to geometric proximity with preference to R_i, while maintaining the disassemblability of P_i in $[P_i, R_i]$; see Figure 8(i) and 9(d).

A graph design is realized if all above steps succeed. Otherwise, we discard this design and try another one in $\mathbf{C}_i$. If all candidates in $\mathbf{C}_i$ fail, we backtrack to the other nodes in the construction tree following the procedure in Algorithm 1.

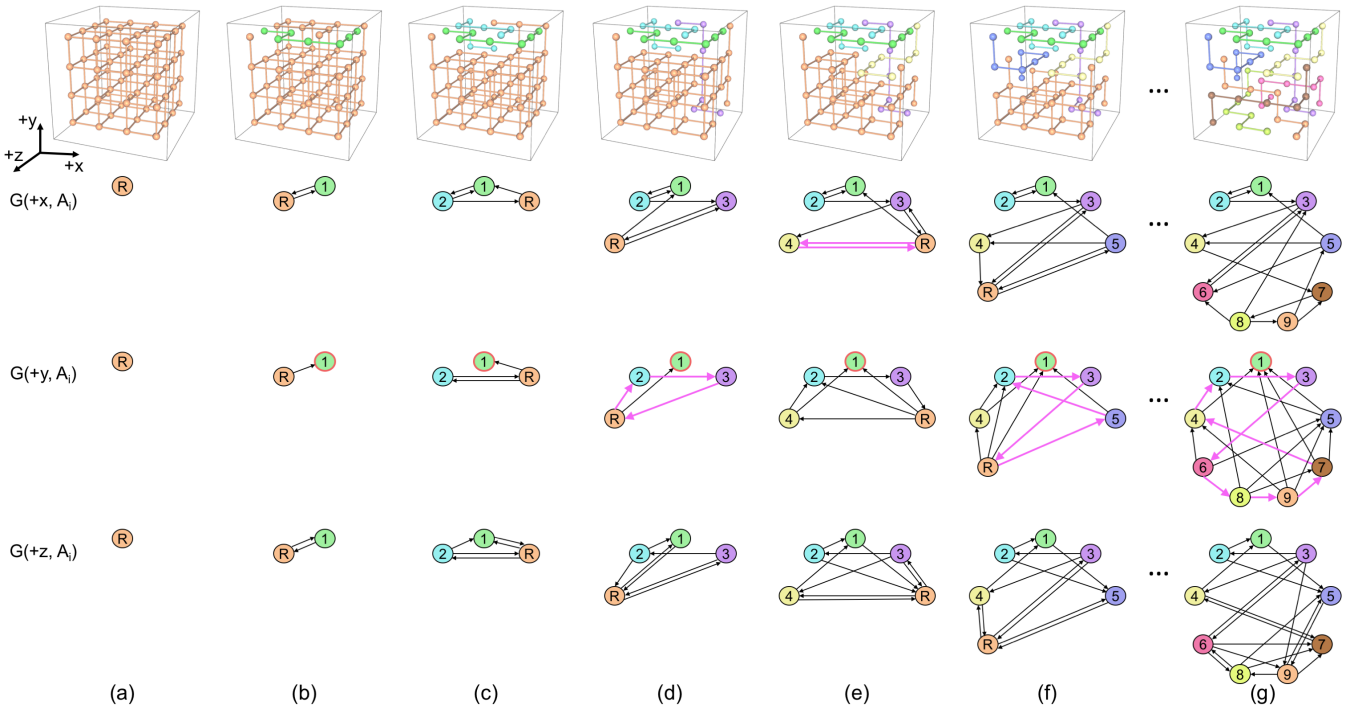


Fig. 10. (a) Starting from a 4×4×4 voxel grid, (b-g) our framework iteratively constructs 9 parts of an interlocking Cube. Three DBGs are drawn for each intermediate assembly at the bottom. Our approach allows immobilizing P_i and R_i by constructing cycles of various sizes (examples colored in purple).

5 RESULTS AND DISCUSSION

In this section we show how our framework can be used to design various kinds of interlocking assemblies, with example applications as puzzles, furniture, sculptures, or architectural designs. We highlight differences to previous approaches to show how our method improves the state of the art and enables new kinds of interlocking assemblies not possible before. For more detailed comparisons and results, we refer to the supplementary material.

5.1 Interlocking Voxelized Structures

Given a voxelized shape and a desired number of parts N as input, our goal here is to decompose the voxel set into a collection of parts that form an interlocking assembly [Song et al. 2012]. Figure 10 shows our iterative design process for creating a 9-part $4 \times 4 \times 4$ interlocking Cube.

The major difference between our approach and [Song et al. 2012] is the graph design of P_i and R_i to ensure interlocking of $\mathbf{A}_i$. Our approach makes use of all previous parts $\{P_1, ..., P_{i-1}\}$ to immobilize P_i and R_i. (i.e., form a cycle in each $\{G(d, A_i)\}$; see Figure 7 and 10), while [Song et al. 2012] only relies on P_{i-1} to immobilize P_i and R_i (i.e., form a 2-part or a 3-part cycle in the DBGs; see Figure 11). Note that our approach can easily generate recursive interlocking puzzles as [Song et al. 2012] by constraining our graph design as shown in Figure 11.

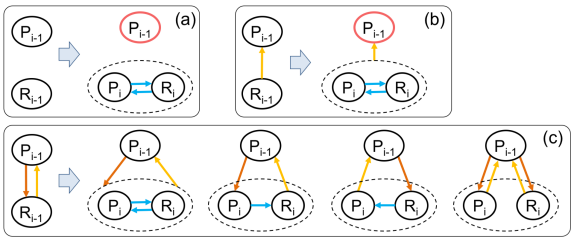


Fig. 11. Illustration of the model of [Song et al. 2012] based on our DBG-based representation. Their approach achieves global interlocking of A_n by requiring every $[P_{i-1}, P_i, R_i]$ $(2 \le i \le n)$ to form a local interlocking group with P_{i-1} as the key. In detail, P_{i-1} and R_{i-1} in $G(d, A_{i-1})$ are possible to have (a) zero, (b) one, and (c) two directed edges. (a&b) P_i and R_i are immobilized in a 2-part cycle $[P_i, R_i]$; (c) P_i and R_i are immobilized in either a 2-part cycle $[P_i, R_i]$, $[P_{i-1}, P_i]$, $[P_{i-1}, R_i]$, or a 3-part cycle $[P_{i-1}, P_i, R_i]$.

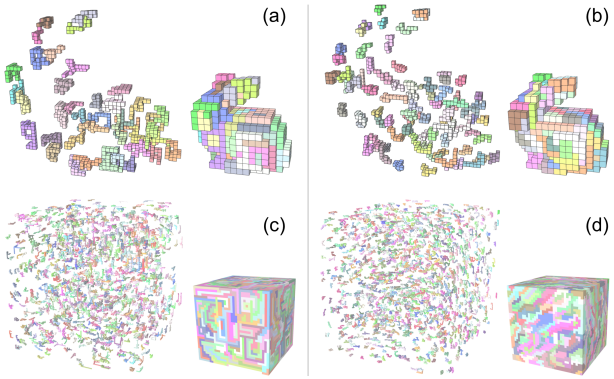


Fig. 12. (a&b) Interlocking Bunnies (966 voxels). (a) The method of [Song et al. 2012] can find an assembly with maximally 40 parts while (b) our approach can find one with 80 parts. (c&d) Interlocking $35 \times 35 \times 35$ Cubes. (c) The method of [Song et al. 2012] can find an assembly with maximally 1250 parts while (d) our approach can find one with 1500 parts.

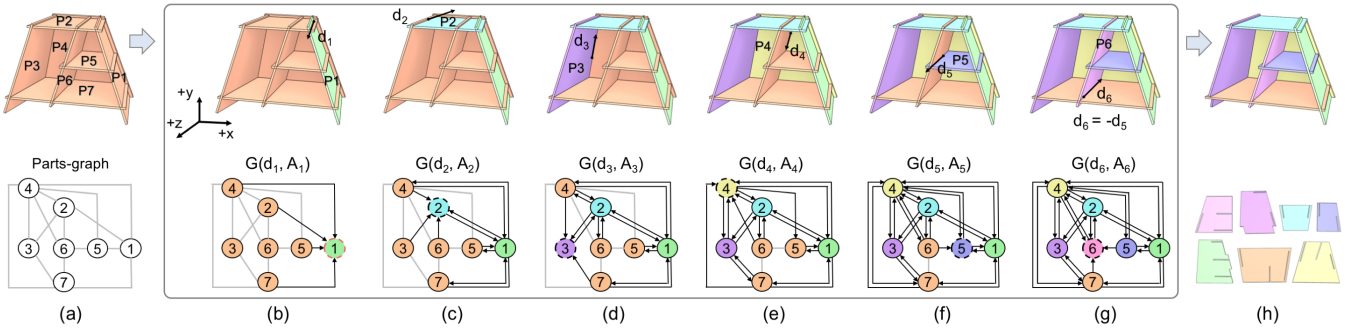


Fig. 13. Design of a 7-part interlocking Cabinet by our approach. (a) Input design and parts-graph. (b-g) The iterative procedure to plan the joints, where the removal direction d_i of P_i is shown in the top row and the active DBG $G(d_i, A_i)$ is shown at the bottom row. All orange nodes in each DBG form R_i and the node with dashed boundary is P_i. (h) Interlocking result and the corresponding parts.

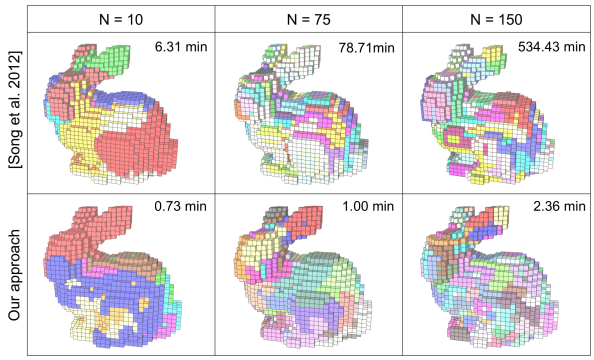


Fig. 14. Our approach is much faster than [Song et al. 2012] when generating 10-, 75-, and 150-part $30 \times 29 \times 23$ Bunnies.

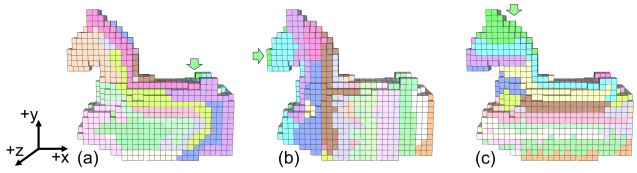
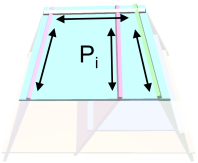


Fig. 15. Design a 20-part Isidore Horse with different criteria for ranking $\{\mathbf{A}_{i+1}\}$: (a) compactness of R_{i+1} (default criteria); (b) X-coordinate of P_{i+1}; and (c) Y-coordinate of P_{i+1}. The ranking criteria could affect the assembly sequence; e.g., parts are disassembled from left to right for (b) and from top to bottom for (c), starting from the green key part (see the green arrow).

Exploiting all existing blocking relations to immobilize P_i and R_i provides significantly more degrees of freedom for designing interlocking assemblies. This allows us to incorporate additional design goals, for example, on object appearance as for the Cartoon Dog in Figure 1. Here we impose constraints that avoid cutting seams across geometric features, so that eyes, ears, nose, and tail are each assigned to a single assembly part. Second, as shown in Figure 12, our approach can find interlocking assemblies with more parts than [Song et al. 2012] for the same input. Given the same input model and the same N, our approach also takes a much shorter time than [Song et al. 2012] to generate results, even though we explore a much richer space of possible assemblies; see Figure 14. Lastly, our approach gives users more control for generating results by allowing them to select their desired criteria for ranking $\{\mathbf{A}_{i+1}\}$; see Figure 15 for an example.

5.2 Interlocking Plate Structures

A second class of assemblies that we can create with our approach are interlocking plate structures that have applications in furniture design or architecture, for example. These assemblies differ in two main ways from the voxelized assemblies described above. First, the geometry of parts and their connections are predefined. We model part connections with an undirected *parts-graph*, in which nodes represent parts, and edges connect two contacting/intersecting parts; see Figure 13(a). Blocking relations can only be constructed between parts connected in the parts-graph. The dual of a *parts-graph* is a *joints-graph*, where nodes represent joints and edges represent parts; see Figure 19(a). Second, we use a set of predefined joint geometries to impose blocking relations between each pair of adjacent parts in the structure. Specifically, we consider mortise-and-tenon, halved, and dovetail joints that restrict each part to move along a single direction; see Figure 2. To support non-orthogonal part connections, we consider suitable variants of mortise-and-tenon and halved joints; see Figure 16. In plate structures, the edge vectors shared between each part P_i and its adjacent parts indicate the base directions $\{d\}$ (see the arrows in the inset for examples), which degenerate into six axial directions for structures where parts are orthogonally connected; see Figure 17(a).



To address the above specifics of interlocking plate structures, we have the following adaptations compared to the voxelized structures. First, instead of decomposing R_{i-1} into P_i and R_i, we iteratively select a single part P_i from the input, and consider the set of unselected parts as R_i; see Figure 13(b-g). Second, rather than constructing geometry of P_i and R_i, we construct joints between P_i and each part in R_i that are connected with P_i in the parts-graph denoted as $R_i^{'}$ such that $\mathbf{A}_i$ ($i \geq 2$) is interlocking. Since all our employed joints allow a single removal direction of the parts, the removal direction of P_i in $[P_i, R_i]$ denoted as d_i completely defines the joints to be constructed

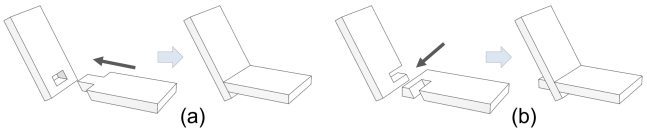


Fig. 16. Variants of (a) mortise-and-tenon joints and (b) halved joints that support non-orthogonal part connections with surface contact.

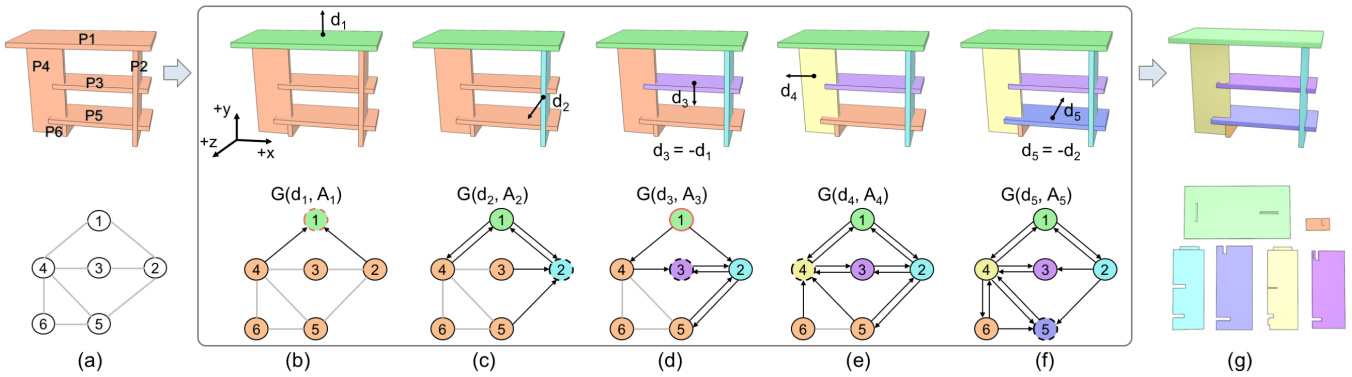


Fig. 17. Design of a 6-part interlocking Table with orthogonal joints. (a) Input design and parts-graph. (b-f) The iterative procedure to plan the joints. (g) Interlocking result and the corresponding parts.

beween P_i and each part in $R_i^{'}$. Third, to achieve interlocking, we only need to ensure that the *active base DBG* $G(d_i, A_i)$ is strongly connected; see DBGs in Figure 13(b-g). The other base DBGs should remain strongly connected since the newly introduced joints only allow part moving along d_i but not the other base directions.

Our iterative approach is detailed as follows:

- *Iterative Design Framework.* Starting from the key P_1, we iteratively select a single part P_i from parts in R_{i-1}. We avoid selecting P_i that is a cut point in the remaining parts-graph of R_{i-1} to keep the geometry of R_i connected. Once P_i is selected, our task is to select d_i from the edge vectors $\{e_i\}$ shared between P_i and its adjacent parts.
- *Generating the key.* Generally, we select P_1 as the part with the most parallel edge vectors, and use this direction as P_1's removal direction d_1 to facilitate joint construction on the key; see Figure 13(b). This is because we create a halved joint for the edge that is parallel to d_1, a mortise-tenon joint for the edge that is nearly perpendicular to d_1 (angle within $[45°, 135°]$), and an empty joint for the other edges. After selecting P_1 and d_1, we plan a joint between P_1 and each part in $R_1^{'}$ (e.g., $R_1^{'} = \{P_2, P_4, P_5, P_7\}$ in Figure 13(b)) such that P_1 is only movable along d_1.
- *Generating P_i and R_i ($i > 1$).* At the graph design stage, we need to select d_i from $\{e_i\}$ such that $G(d_i, A_i)$ is strongly connected, which can be classified into two cases. The first case is that $\pm d_i \notin \{d_1, ..., d_{i-1}\}$. For this case, we build $G(d_i, A_i)$ by converting each undirected edge among $\{P_1, ..., P_{i-1}, R_{i-1}\}$ in the parts-graph into two directed edges and adding a single directed edge between P_i and each part in $R_i^{'}$. The $G(d_i, A_i)$ should be strongly connected by default; see Figure 13(c). The second case is that d_i or $-d_i \in \{d_1, ..., d_{i-1}\}$, say $d_i = d_k$ $(1 \le k \le i-1)$. $G(d_i, A_i)$ inherits all blocking directions from $G(d_k, A_{i-1})$ and we add a single directed edge between P_i and each part in $R_i^{'}$. We try each of $\pm d_i$ and accept d_i $(-d_i)$ if $G(d_i, A_i)$ $(G(-d_i, A_i))$ is strongly connected; see Figure 13(g).

  At the geometry realization stage, we use constructive solid geometry to create the joint geometry on P_i and each part in $R_i^{'}$ according to the joint type planned at the graph design stage. We rank the resulting candidates of $\mathbf{A}_i$ in ascending order of the number of empty joints in $\mathbf{A}_i$.

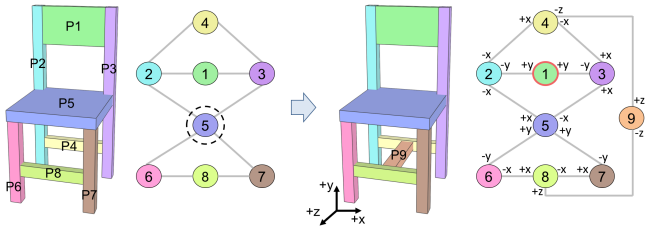


Fig. 18. Left: a Chair and its parts-graph, where a cut point (i.e., P_5) exists. Right: after adding a new part (i.e., P_9), our approach can generate an interlocking joint configuration, where the axial removal direction allowed by each joint is shown in the corresponding edge in the parts-graph.

Figure 13(h) shows an interlocking Cabinet designed by our approach. Besides furniture, plate structures also can be used to approximate a free form shape; see the 33-part Lizard in Figure 1 for an example. Since the DBG-based approach is not sufficient to test interlocking for these plate structures with non-orthogonal part connections (see Subsection 3.2), we verify the two results by running the inequality-based interlocking test and find that both results pass the test.

In the special case of orthogonal joints, our approach generalizes the furniture design work of [Fu et al. 2015]; see Figure 17 for an example. In particular, Fu et al. [2015] focus on furniture with 3- or 4-part cyclic substructures since their approach requires these substructures to construct LIGs. In contrast, our approach does not have such a limitation; see the Bookshelf with four 6-part cyclic substructures in Figure 1.

Lastly, inspired by our DBG-based representation, we find that a parts-graph with a cut point cannot be interlocking, no matter what kinds of joints are used; see Figure 18(left) for an example and supplementary material for a proof. This observation allows us to modify a given input to make it possible to be interlocking by adding a minimal number of new parts in the parts-graph in order to remove the cut point; see Figure 18(right) for an example.

5.3 Interlocking Frame Structures

As a new class of interlocking assembly, we propose *interlocking frame structures* that can be considered as a hybrid of the voxelized and the plate assemblies. A frame structure is a network of beams joined to represent a desired target shape. As input we assume a

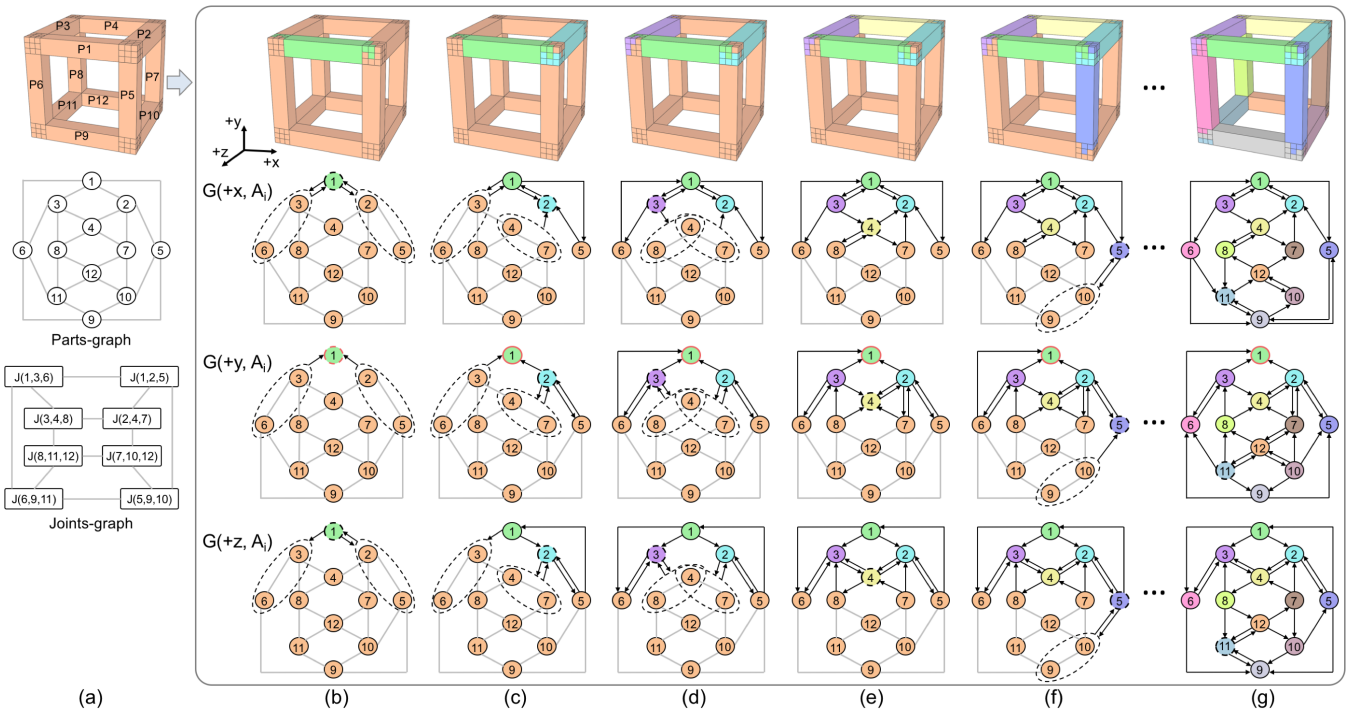


Fig. 19. Design of a 12-part interlocking Frame Cube. (a) Input frame design, parts-graph, and joints-graph. (b-f) The iterative procedure to construct the voxel joints, where the dashed ellipses highlight the set of parts in R_i that connect with P_j ($j \leq i$) using voxel joints. (g) The interlocking result.

3D polygonal mesh, where each edge represents a beam and each vertex represents a joint.

Compared with the plate assemblies, frame structures have two more challenges. First, frame structures require connecting more than two parts at a joint, making traditional woodworking joints unsuitable; see the joints-graph in Figure 19(a). Thus, we propose to connect the beams with cube-shaped *voxel joints*. To make the problem tractable, we assume that each face of the joint connects to at most one beam at the center voxel, and thus each joint connects at most six beams (i.e., valence of the input mesh should be at most 6). Second, we need to individually optimize the geometry of each voxel joint. We place an axis-aligned $3 \times 3 \times 3$ cube at each joint location and partition the cubes into partial joints to restrict the relative movement of the connected beams; see the eight corners of the Frame Cube in Figure 19(a). Compared to the plate structures, these specifics require the following adaptations to our framework:

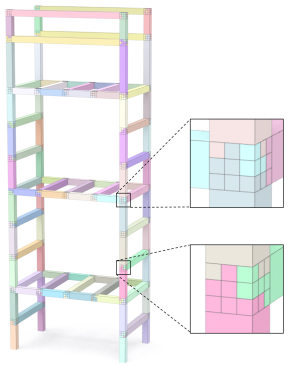
- *Generating the key.* For the voxel joint at each end of P_1, we take the voxel preassigned to P_1 as a seed and include more voxels to P_1 such that it is movable along a singe axial direction following Subsection 4.2. Denote the subset of parts that connect with P_1 at each end as $S_1^k = \{P_l\}$, where $k \in \{1, 2\}$; see the dashed circles in Figure 19(b). After constructing P_1, we draw directed edges between P_1 and each S_1^k in the DBGs accordingly.
- *Generating P_i and R_i ($i > 1$).* At the graph design stage, we classify parts in each S_i^k into two groups: $\{\grave{P}_l\}$ where each part is from $\{P_1, ..., P_{i-1}\}$ and $\{\acute{P}_l\}$ where each part is from R_i. If $\{\acute{P}_l\}$ is empty, the blocking relations within this voxel joint are completely defined. So the graph design of P_i and R_i can be skipped for this joint; see joint $J(1, 2, 5)$ in Figure 19(f), where $P_i = P_5$, $\{\grave{P}_l\} = \{P_1, P_2\}$. If $\{\grave{P}_l\}$ is empty, we do not need to distribute external blocking relations in this joint; see joint $J(2, 4, 7)$ in Figure 19(c), where $P_i = P_2$, $\{\acute{P}_l\} = \{P_4, P_7\}$. Otherwise, we distribute external blocking relations associated with $\{\grave{P}_l\}$ to P_i and parts in $\{\acute{P}_l\}$ respectively; see joint $J(1, 2, 5)$ in Figure 19(c). Constructing internal blocking relations is restricted to P_i and parts in $\{\acute{P}_l\}$ for each S_i^k. To ensure that P_i is disassemblable in $[P_i, R_i]$, say along d_i, we construct a single directed edge between P_i and each part in $\{\acute{P}_l\}$ for both S_i^k in at least one DBG; see $G(+x, A_2)$ in Figure 19(c) for an example.

The geometry realization of P_i and R_i is conducted within the voxel joint at each end of P_i following the approach in Subsection 4.3; see corners of the Frame Cube in Figure 19(c-f).

Fig. 20. Interlocking $1.0m \times 0.5m \times 0.5m$ Frame Chair. The voxel joints are fabricated by gluing wooden cubes in the spatial arrangement computed by our algorithm. When attached to the corresponding wooden pillars, all pillars can be connected into a steady interlocking assembly.

Figure 19(g) shows the resulting interlocking Frame Cube, in which every three beams joining at a corner are connected by a carefully constructed three-way voxel joint. Note that all joints are distinct even though all corners are symmetric. This illustrates how the assembly order dictates the geometry of the joints, more so than the geometry of the parts.

Our approach can generate frame structures with different joint valence, e.g., the valence in the Frame Chair in Figure 20 can be 2, 3 or 4. We fabricate this result using wooden pillars and small wooden cubes to validate its steadiness; see supplementary video for the live demo. The Flower in Figure 1 shows another result, where curved beams are connected by the voxel joints to form an appealing structure. Our approach can generate frame structures with a large number of parts; see the 92-part Scaffold in the inset. To the best of our knowledge, these are the first *single-key* interlocking frame structures.



5.4 Implementation and Performance

Our C++ implementation runs on an iMac with a 4.2GHz CPU and 32GB memory. In general, the timing performance depends on the input model, the number of parts N, and additional design requirements; see Table 1. The computation time to create interlocking voxelized structures highly depends on the number of desired parts. For example, the interlocking $4 \times 4 \times 4$ Cubes (Figure 10) with 7, 8, and 9 parts take 0.3 seconds, 12 seconds, and 1.13 hours respectively. For larger N, it becomes increasingly difficult to find an interlocking assembly since smaller parts have fewer potential blocking contacts. Enforcing additional design requirements can also increase the computation time substantially. For example, creating Cartoon Dog (Figure 1) without constraints takes 23.3 seconds, while incorporating the appearance constraints increases computation to 1.06 hours, since significantly more backtracking is required to ensure that the constructed parts align with the features. For more details on the implementation of our approach, we refer to the source code provided in the supplementary material.

Table 1. Statistics of results generated by our framework. The labels in 4rd to 6th columns refer to the number N of parts, the number M of base directions, and the time for generating the result.

	Fig.	Model	N	M	Time
Voxelized Structure	1	Cartoon Dog	14	3	1.06 hour
	10	4x4x4 Cube	9	3	1.13 hour
	12	Bunny	80	3	0.74 hour
	12	35x35x35 Cube	1500	3	3.33 hour
Plate Structure	1	Lizard	33	45	2.73 sec
	1	Bookshelf	12	3	0.30 sec
	13	Cabinet	7	5	0.09 sec
	17	Table	6	3	0.02 sec
	18	Chair	8	3	0.22 sec
Frame Structure	1	Flower	23	3	4.10 sec
	19	Frame Cube	12	3	0.53 sec
	20	Frame Chair	11	3	15.66 sec
	Inset	Scaffold	92	3	26.41 sec

Our approach creates interlocking plate structures very efficiently due to the relatively small search space. For example, it takes 0.07 seconds to compute Cabinet in Figure 13. Due to the non-orthogonal part connections in the Cabinet, it further takes 0.02 seconds to verify interlocking by performing the inequality-based test. Hence, the total time to generate this result is 0.09 seconds; see Table 1. The other result with non-orthogonal part connections (i.e., Lizard in Figure 1) takes 0.03 seconds to verify its interlocking. Designing frame structures is also fast since the $3 \times 3 \times 3$ cubes that are optimized at each joint location have only 27 voxels. In particular, it takes much longer time to generate the Frame Chair than the Frame Cube although they have similar number of parts; see Table 1. This is mainly due to the lower number of cycles in the parts-graph of the Frame Chair, which makes it harder to find interlocking configurations.

6 CONCLUSION

Combining parts into an interlocking assembly imposes strong constraints on the part geometry and arrangement. Our novel framework leverages carefully designed graph representations and algorithms to efficiently test whether an assembly of parts is interlocking. This efficiency and the generality of our tree-traversal algorithm allows us to explore a significantly larger configuration space compared to previous solutions. As a consequence, our approach can find interlocking assemblies for models that previous methods fail on, allows integrating additional geometric constraints to better meet the design goals, and enables new types of assemblies not possible before.

Limitations and Future Work. Our work has several limitations that open up interesting directions for future research. First, the DBG-based representation models only the infinitesimal translational motions of the parts. Finding a conceptual representation that supports extended translational motions (e.g., for avoiding global collision) would be helpful for testing interlocking and disassemblability in a unified manner. Currently we assume planar inter-part contact and translational assembly motion. While this simplifies the conceptual representation as well as the fabrication and construction of interlocking assemblies, generalizations to non-planar contact and more complex assembly motions [Zhang et al. 2018] could lead to new types of assemblies. We currently do not analyze the structural implications of the way individual parts are connected. As a future work, it would be valuable to optimize the stress distribution to avoid local stress concentrations in the assembly. Another important aspect that is currently not covered in our work is tolerance handling. Fabrication imprecisions lead to deviations in the part geometries that can accumulate and negatively impact the stability of the assembly. How to design for robustness against such error accumulation is an exciting future research problem. The frame structure that we introduce in this paper is just one instance of a broader class of possible assemblies where joint geometries are optimized together with the assembly, instead of being selected from a set of predefined joint types. Voxelized cube joints do not necessarily provide the most appropriate connection and novel joint

typologies could be discovered in the future that are better suited for the kind of multi-part joints that we studied in this paper. Other potential directions for future work include assemblies of deformable parts [Skouras et al. 2015] or reconfigurable assemblies.

ACKNOWLEDGMENTS

We thank the reviewers for the valuable comments, Zhe Chen and Fengyuan Yang for their help in fabricating the frame chair, and Mina Konaković-Luković for proofreading the paper. This work was supported by the NCCR Digital Fabrication, funded by the Swiss National Science Foundation, NCCR Digital Fabrication Agreement #51NF40-141853.

REFERENCES

Maneesh Agrawala, Doantam Phan, Julie Heiser, John Haymaker, Jeff Klingner, Pat Hanrahan, and Barbara Tversky. 2003. Designing Effective Step-By-Step Assembly Instructions. *ACM Trans. on Graph. (SIGGRAPH)* 22, 3 (2003), 828–837.

Marco Attene. 2015. Shapes In a Box: Disassembling 3D Objects for Efficient Packing and Fabrication. *Computer Graphics Forum* 34, 8 (2015), 64–76.

Lujie Chen and Lawrence Sass. 2015. Fresh Press Modeler: A generative system for physically based low fidelity prototyping. *Computers & Graphics* 54 (2015), 157–165.

Xuelin Chen, Hao Zhang, Jinjie Lin, Ruizhen Hu, Lin Lu, Qixing Huang, Bedrich Benes, Daniel Cohen-Or, and Baoquan Chen. 2015. Dapper: Decompose-and-pack for 3D Printing. *ACM Trans. Graph.* 34, 6 (2015), 213:1–213:12.

Paolo Cignoni, Nico Pietroni, Luigi Malomo, and Roberto Scopigno. 2014. Field-aligned Mesh Joinery. *ACM Trans. on Graph.* 33, 1 (2014), 11:1–11:12.

Fernando de Goes, Pierre Alliez, Houman Owhadi, and Mathieu Desbrun. 2013. On the Equilibrium of Simplicial Masonry Structures. *ACM Trans. Graph. (SIGGRAPH)* 32, 4 (2013). Article 93.

Bansal Deepak. 2012. Sustainable Dry Interlocking Block Masonry Construction. In *15th International Brick and Block Masonry Conference.*

Mario Deuss, Daniele Panozzo, Emily Whiting, Yang Liu, Philippe Block, Olga Sorkine-Hornung, and Mark Pauly. 2014. Assembling Self-Supporting Structures. *ACM Trans. on Graph. (SIGGRAPH Asia)* 33, 6 (2014). Article No. 214.

Noah Duncan, Lap-Fai Yu, and Sai-Kit Yeung. 2016. Interchangeable Components for Hands-On Assembly Based Modelling. *ACM Trans. on Graph. (SIGGRAPH Asia)* 35, 6 (2016). Article No. 234.

Ursula Frick, Tom Van Mele, and Philippe Block. 2015. Decomposing Three-Dimensional Shapes into Self-supporting, Discrete-Element Assemblies. In *Modelling Behaviour.* 187–201.

Chi-Wing Fu, Peng Song, Xiaoqi Yan, Lee Wei Yang, Pradeep Kumar Jayaraman, and Daniel Cohen-Or. 2015. Computational Interlocking Furniture Assembly. *ACM Trans. on Graph. (SIGGRAPH)* 34, 4 (2015). Article No. 91.

Somayé Ghandi and Ellips Masehian. 2015. Review and taxonomies of assembly and disassembly path planning problems and approaches. *Computer-Aided Design* 67-68 (2015), 58–86.

Jianwei Guo, Dong-Ming Yan, ErLi, Weiming Dong, Peter Wonka, and Xiaopeng Zhang. 2013. Illustrating the disassembly of 3D models. *Computers & Graphics* 37, 6 (2013), 574–581.

Jingbin Hao, Liang Fang, and Robert E. Williams. 2011. An Efficient Curvature-based Partitioning of Large-scale STL Models. *Rapid Prototyping Journal* 17, 2 (2011), 116–127.

Kristian Hildebrand, Bernd Bickel, and Marc Alexa. 2012. crdbrd: Shape Fabrication by Sliding Planar Slices. *Comp. Graph. Forum* 31, 2 (2012), 583–592.

Ruizhen Hu, Honghua Li, Hao Zhang, and Daniel Cohen-Or. 2014. Approximate Pyramidal Shape Decomposition. *ACM Trans. Graph.* 33, 6 (2014), 213:1–213:12.

Bongjin Koo, Jean Hergel, Sylvain Lefebvre, and Niloy J. Mitra. 2017. Towards Zero-Waste Furniture Design. *IEEE Trans. Vis. & Comp. Graphics* 23, 12 (2017), 2627–2640.

Manfred Lau, Akira Ohgawara, Jun Mitani, and Takeo Igarashi. 2011. Converting 3D furniture models to fabricatable parts and connectors. *ACM Trans. on Graph. (SIGGRAPH)* 30, 4 (2011). Article No. 85.

Wilmot Li, Maneesh Agrawala, Brian Curless, and David Salesin. 2008. Automated Generation of Interactive 3D Exploded View Diagrams. *ACM Trans. on Graph. (SIGGRAPH)* 27, 3 (2008). Article No. 101.

Yang Liu, Pan Hao, John Snyder, Wenping Wang, and Baining Guo. 2013. Computing Self-Supporting Surfaces by Regular Triangulation. *ACM Trans. Graph. (SIGGRAPH)* 32, 4 (2013). Article 92.

Linjie Luo, Ilya Baran, Szymon Rusinkiewicz, and Wojciech Matusik. 2012. Chopper: Partitioning Models into 3D-printable Parts. *ACM Trans. Graph.* 31, 6 (2012), 129:1–129:9.

James McCrae, Nobuyuki Umetani, and Karan Singh. 2014. FlatFitFab: Interactive Modeling with Planar Sections. In *ACM Symposium on User Interface Software and Technology.* 13–22.

Masaaki Miki, Takeo Igarashi, and Philippe Block. 2015. Parametric Self-supporting Surfaces via Direct Computation of Airy Stress Functions. *ACM Trans. Graph. (SIGGRAPH)* 34, 4 (2015). Article 89.

Daniele Panozzo, Philippe Block, and Olga Sorkine-Hornung. 2013. Designing Unreinforced Masonry Models. *ACM Trans. on Graph. (SIGGRAPH)* 32, 4 (2013). Article No. 91.

Ronald Richter and Marc Alexa. 2015. Beam meshes. *Computers & Graphics* 53 (2015), 28–36.

Matthias Rippmann, Tom Van Mele, Mariana Popescu, Edyta Augustynowicz, Tomás Méndez Echenagucia, Cristián Calvo Barentin, Ursula Frick, and Philippe Block. 2016. The Armadillo Vault: Computational design and digital fabrication of a freeform stone shell. In *Advances in Architectural Geometry.* 344–363.

Yuliy Schwartzburg and Mark Pauly. 2013. Fabrication-aware Design with Intersecting Planar Pieces. *Comp. Graph. Forum* 32, 2 (2013), 317–326.

Tianjia Shao, Dongping Li, Yuliang Rong, Changxi Zheng, and Kun Zhou. 2016. Dynamic Furniture Modeling Through Assembly Instructions. *ACM Trans. on Graph. (SIGGRAPH Asia)* 35, 6 (2016). Article No. 172.

Mélina Skouras, Stelian Coros, Eitan Grinspun, and Bernhard Thomaszewski. 2015. Interactive Surface Design with Interlocking Elements. *ACM Trans. on Graph. (SIGGRAPH Asia)* 34, 6 (2015). Article No. 224.

A. S. Solodovnikov. 1979. *Systems of Linear Inequalities.* Mir Publishers.

Peng Song, Bailin Deng, Ziqi Wang, Zhichao Dong, Wei Li, Chi-Wing Fu, and Ligang Liu. 2016. CofiFab: Coarse-to-Fine Fabrication of Large 3D Objects. *ACM Trans. on Graph. (SIGGRAPH)* 35, 4 (2016). Article No. 45.

Peng Song, Chi-Wing Fu, and Daniel Cohen-Or. 2012. Recursive Interlocking Puzzles. *ACM Trans. on Graph. (SIGGRAPH Asia)* 31, 6 (2012). Article No. 128.

Peng Song, Chi-Wing Fu, Yueming Jin, Hongfei Xu, Ligang Liu, Pheng-Ann Heng, and Daniel Cohen-Or. 2017. Reconfigurable Interlocking Furniture. *ACM Trans. on Graph. (SIGGRAPH Asia)* 36, 6 (2017). Article No. 174.

Peng Song, Zhongqi Fu, Ligang Liu, and Chi-Wing Fu. 2015. Printing 3D Objects with Interlocking Parts. *Comp. Aided Geom. Des.* 35-36 (2015), 137–148.

Rob Stegmann. 2018. Rob's Puzzle Page - Interlocking Puzzles. (2018). http://www.robspuzzlepage.com/interlocking.htm.

Alan Song-Ching Tai. 2012. *Design For Assembly: A Computational Approach to Construct Interlocking Wooden Frames.* Ph.D. Dissertation. Massachusetts Institute of Technology.

Chengcheng Tang, Xiang Sun, Alexandra Gomes, Johannes Wallner, and Helmut Pottmann. 2014. Form-finding with polyhedral meshes made simple. *ACM Trans. Graph. (SIGGRAPH)* 33, 4 (2014). Article 70.

Robert Tarjan. 1972. Depth-first search and linear graph algorithms. *SIAM J. Comput.* 1, 2 (1972), 146–160.

J. Vanek, J. A. Garcia Galicia, B. Benes, R. Měch, N. Carr, O. Stava, and G. S. Miller. 2014. PackMerger: A 3D Print Volume Optimizer. *Computer Graphics Forum* 33, 6 (2014), 322–332.

E. Vouga, M. Höbinger, J. Wallner, and H. Pottmann. 2012. Design of self-supporting surfaces. *ACM Trans. Graph. (SIGGRAPH)* 31, 4 (2012), Article 87.

Emily Whiting, John Ochsendorf, and Frédo Durand. 2009. Procedural Modeling of Structurally-Sound Masonry Buildings. *ACM Trans. Graph. (SIGGRAPH Asia)* 28, 5 (2009). Article 112.

Randall H. Wilson. 1992. *On Geometric Assembly Planning.* Ph.D. Dissertation. Stanford University.

Randall H. Wilson and Jean-Claude Latombe. 1994. Geometric Reasoning About Mechanical Assembly. *Artificial Intelligence* 71, 2 (1994), 371–396.

Shi-Qing Xin, Chi-Fu Lai, Chi-Wing Fu, Tien-Tsin Wong, Ying He, and Daniel Cohen-Or. 2011. Making Burr Puzzles from 3D Models. *ACM Trans. on Graph. (SIGGRAPH)* 30, 4 (2011). Article No. 97.

Jiaxian Yao, Danny M. Kaufman, Yotam Gingold, and Maneesh Agrawala. 2017b. Interactive Design and Stability Analysis of Decorative Joinery for Furniture. *ACM Trans. on Graph.* 36, 2 (2017). Article No. 20.

Miaojun Yao, Zhili Chen, Weiwei Xu, and Huamin Wang. 2017a. Modeling, Evaluation and Optimization of Interlocking Shell Pieces. *Comp. Graph. Forum* 36, 7 (2017), 1–13.

Yinan Zhang and Devin Balkcom. 2016. Interlocking Structure Assembly with Voxels. In *IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems.* 2173–2180.

Yinan Zhang, Emily Whiting, and Devin Balkcom. 2018. Assembling and Disassembling Planar Structures with Divisible and Atomic Components. *IEEE Transactions on Automation Science and Engineering* 15, 3 (2018), 945–954.