Procedural bridges-and-pillars support generation

M. Freire, S. Hornus, S. Perchy and S. Lefebvre

Université de Lorraine, CNRS, Inria, LORIA, F-54000 Nancy, France

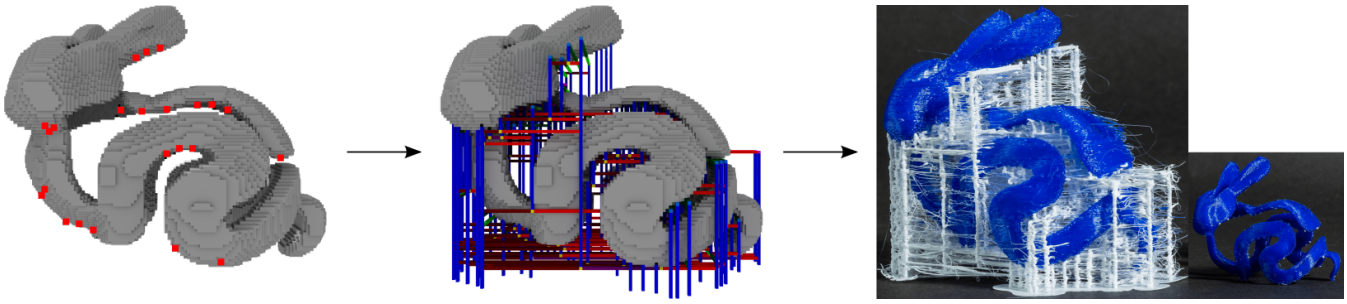


Figure 1: *Our algorithm generates bridges-and-pillars supports in a voxel grid surrounding an object, with complexity independent from the number of points to support. The support graph is converted into geometry, sliced and 3D printed. Our supports carefully avoid the part, leaving no unnecessary scars (model from [SU14]). Thin white threads are due to optimizing away retraction moves, but none touch the part.*

Abstract

Additive manufacturing requires support structures to fabricate parts with overhangs. In this paper, we revisit a known support structure based on bridges-and-pillars (see Figure 1). The support structures are made of vertical pillars supporting horizontal bridges. Their scaffolding structure makes them stable and reliable to print. However, the algorithm heuristic search does not scale well and is prone to produce contacts with the parts, leaving scars after removal.

We propose a novel algorithm for this type of supports, focusing on avoiding unnecessary contacts with the part as much as possible. Our approach builds upon example-based model synthesis to enable early detection of collision-free passages as well as non-reachable regions.

CCS Concepts

• ***Applied computing*** → ***Computer-aided design;*** • ***Computing methodologies*** → ***Shape modeling;***

1. Introduction

Many 3D printing processes can only stack a new layer on top of an already fabricated surface. Printing features in overhang thus requires disposable support structures, cleaned after fabrication. They must be easy to remove and most importantly, touch the object only where strictly necessary to avoid scarring.

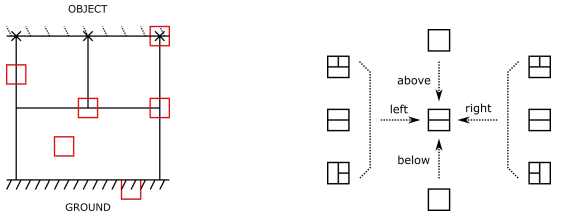
Many support techniques have been proposed over the years, please refer to [LEM*17; LGC*18; JXS18]. In this work we revisit the generation algorithm of the bridges-and-pillars support structures of [DHL14], which is available in the *IceSL* [INR13] slicing software. This technique relies on the bridging capability of FDM printers to produce a scaffolding geometry that is stable and prints reliably.

While effective, the heuristic 'next bridge' search proposed in [DHL14] suffers two main drawbacks. First, in cramped geometries the algorithm struggles to detect collision-free bridges. Most notably, it cannot detect narrow passages to go through. Besides, collision checking is expensive and not implemented in the publicly available version in the *IceSL* software. Second, it scales with the fourth power of the number of points to support, making it impractical for large models.

We propose a novel algorithm addressing the aforementioned drawbacks. We build upon the *example-based model synthesis* technique [Mer07; Gum16] that generates geometry from a given example. Model synthesis draws inspiration from general *constraint satisfaction problems* algorithms such as AC3 [RRN19]. It works in a discrete voxel space where each voxel is given a label.

Labels give geometric meaning to the voxel they are attached to. In our approach, depending on its label a voxel can represent a

DOI: 10.2312/egs.20221025



(a) *Side view of a support, red squares represent different labels.* **(b)** *Side view, possible adjacent labels of a bridge element label.*

Figure 2: *Label geometries and adjacency constraints.*

bridge or a pillar element, a junction, a point to support (*anchor*), or a part of the object. Within each of those elements, multiple labels are used to represent different roles in the support. Choosing a label for a voxel constrains its surrounding voxels. For example, a bridge element voxel cannot be fully surrounded by empty voxels, since bridges should be supported at both ends. Similarly, a voxel needing support cannot have an empty voxel below it. This information is represented by a set of *adjacency constraints* that determines if two labels — or more accurately, the voxels they are assigned to — can be adjacent to each other in a specific direction. This is illustrated in figure 2.

Algorithm 1: Model (M) synthesis algorithm

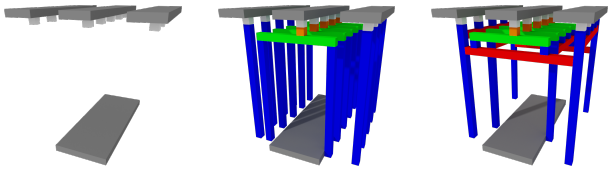
Input: obj: object to print and points needing support
L, AC: set of labels, adjacency constraints
Data: $M(v) \in L$: label at voxel v; if unassigned, $M(v) = \perp$
$A(v) \subseteq L$: labels allowed for voxel v
Function `synthesize(obj)`
  $M(v) \leftarrow \perp$ for all voxels
  `initModel(obj, M)`
  $U \leftarrow$ set of unassigned voxels in M
  while U *is not empty* **do**
    choose $v \in U$, choose $l \in A(v)$
    $M(v) \leftarrow l$
    `/* update allowed labels */`
    `propagateConstraints(A, v, AC)`
  end

The outline of the model synthesis process [Mer07; Gum16] is given in algorithm 1. We highlighted in blue the elements we modify: the constraints, which unassigned voxel is processed next, and how its label is chosen based on adjacency constraints. Constraint propagation is explained in depth in [Mer07].

Our main contribution is the design of a specially crafted set of constraints and a custom next voxel and label selection, that together synthesize an initial structure minimizing contact with the part *without trial-and-error*. This is in stark contrast to standard synthesis algorithms which often encounter contradictions where no label is allowed for a voxel, and either backtrack or have to restart.

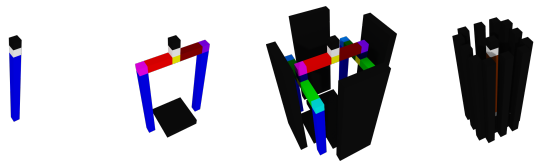
2. Two-phase algorithm

Our algorithm operates in two phases, illustrated in Figure 3 on a toy example. Both phases follow the synthesis process outlined in



(a) *Voxelized object* **(b)** *Phase one* **(c)** *Phase two*

Figure 3: *Two-phase approach. Pillars in blue, tables in green, bridges in red, object in gray, anchors in white.*



(a) *Pillar–ground* **(b)** *Single table* **(c)** *Double table* **(d)** *Pillar–object*

Figure 4: *Structures (color) generated by the first phase from an anchor (white) around the object (black).*

Algorithm 1, with different sets of labels and adjacency constraints. The first one builds tables and isolated pillars. Its goal is to generate the simplest type of structure to support every anchor while *minimizing* the number of pillars standing on the object. The second phase uses the result of the first phase as a starting point and optimizes the generated structure by building bridges between isolated pillars, thus reducing its overall length.

The type of structures generated by the algorithm is described by two example models, one for each phase. These models define the adjacency constraints: if two labels are not adjacent in the example, they cannot be adjacent in the output. Both example models are designed in such a way to allow bridge-and-pillar support structures. The two phases use different — albeit strongly related — example models.

2.1. First phase: object avoidance

Anchors (i.e. points needing support) that can be seen from the ground when looking at the model from the bottom can be easily supported with a single vertical pillar. Of course many anchors are not visible in this way, since the object itself can obstruct the vertical line of sight. Our example model leads to the synthesis of different structures avoiding the object, illustrated in Figure 4 for simple scenes.

These are, by decreasing priority: a) an isolated pillar standing on ground, b) a single table, c) a double table or d) an isolated pillar standing on the object. The example model for the first phase is shown in Figure 5. Different colors correspond to different labels and different pieces of the support geometry (see also Figure 2). There are in fact more labels than there are visible colors due to the

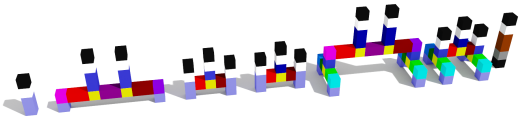


Figure 5: *Example model used in the first phase.*

limited palette. The hierarchical nature of the structures, as well as the use of different labels at each of their levels causes many labels to be removed during each constraint propagation step.

Due to the design of our example model, the initial constraint propagation step after the object and anchors are loaded (`initModel` in Algorithm 1) is actually sufficient to determine how to support each anchor. After propagation, the allowed labels below an anchor already indicate the simplest structures that can support it.

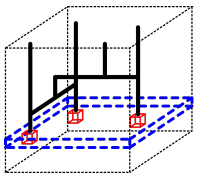
This property is specific to our example model. In fact, with only slightly more permissive example models constraint propagation after assigning a label impacts only a small neighborhood: many labels remain possible even in the vicinity of already assigned voxels. This then requires many iterations to refine the result, and often leads to contradictions. By quickly eliminating many impossible outcomes, our example models enable fast and reliable synthesis.

2.2. Second phase: support optimization

The first phase is optimized to avoid contacts, and does not attempt to reduce the support size. In particular it always generates isolated pillars under unobstructed anchors. Phase two improves the structure by connecting multiple aligned pillars with bridges. This is done by removing all isolated pillars from the result, and then regenerating the structure with the synthesis algorithm, using bridges wherever possible. Compared to the first phase, the example model includes additional structures to allow for these improvements.

3. Choice and assignment heuristics

The order in which unassigned voxels are chosen is crucial to ensure that the algorithm avoids inconsistencies. In an incomplete model, let us choose a voxel far from the already assigned voxels. That voxel would spawn a second structure in that region of the model. If done multiple times, we create many local structures that will all have to meet at some point — and are very likely to disagree.



To solve this, we use a context-dependent order. The algorithm operates slice by slice, from top to bottom (see inset). Within a slice (in blue), voxels whose "up stairs neighbors" are part of the support structure are tagged as seeds (in red). Voxels in the current slice are selected by increasing distance to the set of seeds. Once all voxels in a slice have been assigned, the algorithm goes to the next one until the bottom of the model is reached. By doing this, we always select unassigned voxels that are adjacent to already assigned voxels. This drastically decreases the probability of making a choice that will lead to an inconsistency further down the line.

For a given unassigned voxel, a label is chosen by following a set of priority rules. The first rule states that if a voxel can be empty then it should be, encouraging smaller structures. The second rule, used in the first phase, guarantees that the simplest structure is chosen to support a given anchor. Other rules are used to make tables as short as possible and to push bridges upwards, creating a denser structure close to the object.

4. Results

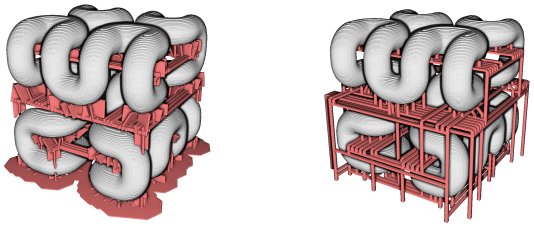
Unless otherwise specified we use a voxel size of 0.5 mm. We printed these models using PLA filament on low-cost Ender3 and CR10 filament printers.

4.1. Implementation

Our algorithm takes a voxel grid as an input and returns a list of segments. The voxels represent the object and the points needing support are explicitly labeled as anchors. The resulting list of segments describes the computed support structure.

Our processing pipeline is given a 3D model and outputs GCode for a physical print. It consists of the following steps: (1) a voxelizer and support point detector, (2) a support structure generator (our method), (3) a support geometry creator and (4) slicing, trajectories and GCode output. The final step is using a standard slicer, while other steps are tailored to our method. (1), (3) and (4) are independent from our method and could be done by other means.

4.2. Comparison with the original algorithm



(a) *Original* (333 *anchors,* 12*s)* **(b)** *Ours* (347 *anchors,* 2*s)*

Figure 6: *Algorithms comparison on the* Hilbert cube *model, between [DHL14] (left) and our algorithm (right).*

We propose a different algorithm to generate structures similar to [DHL14]. Figure 6 compares both on the *Hilbert cube* model. Note how the supports of the original algorithm intersect the object in many places, while ours avoid the object. The support size — filament length for fabrication — is similar for both (2.8m). Here, for a similar number of anchors the execution time of our method is faster (2s vs 12s). The complexity of our approach is independent from the number of anchors. Instead, it increases proportionally to the square of the number of voxels giving it an advantage on larger, more complex models. However, the original can be faster on small models with few anchors. Collision avoidance in [DHL14] is expensive, as each candidate bridge has to be explicitly checked against the model, and the algorithm still often fails to find collision-free solutions. Due to this, collisions avoidance is not even implemented in the publicly available version of [DHL14].

4.3. Result gallery

Table 1 lists execution times for models with varying voxel grid sizes. Also shown is the number of anchors and how many contact points (CP) are created by the algorithm failing to avoid the object. For all models but *knot*, the number of created CP is below 6% of the total number of CP. *knot* has a cramped geometry and the threefold rotational symmetry makes it difficult to generate axis-aligned supports that avoid the object.

Table 1: *Performance for various models, 0.5mm voxels. Computed on an AMD Ryzen 5600X with 16GB of DDR4 RAM.*

Models	Grid dimensions (voxels)	Number of voxels	Execution time (s)	Number of anchors	Created contact points
Gymnast	$81 \times 28 \times 99$	224 532	1	37	2
Hilbert	$70 \times 70 \times 65$	318 500	2	347	0
Knot	$96 \times 101 \times 72$	698 112	4	199	33
Bunny peel	$119 \times 92 \times 104$	1 138 592	9	235	1
Minotaur	$126 \times 94 \times 203$	2 404 332	19	392	16
Enterprise	$318 \times 149 \times 72$	3 411 504	28	1475	0
Fox	$137 \times 166 \times 218$	4 957 756	34	73	0
Thigh left	$138 \times 390 \times 167$	9 987 940	78	3498	43
Cellular thing	$294 \times 286 \times 248$	20 852 832	185	2369	33

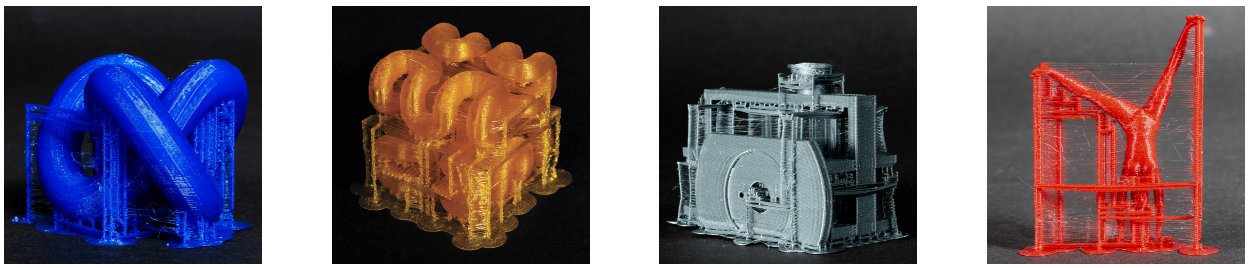


Figure 7: *Models printed with our technique. From left to right, top to bottom: Knot, Hilbert cube, Servo support, Gymnast.*

In Figure 7, note how the supports avoid touching the part. For instance, despite the intricate shape of the *Hilbert cube* (yellow), no pillars are contacting downwards with the print. The same is true of the *servo support* (gray), where multiple horizontal bridges can be seen going through the lateral hole. This avoidance comes at no extra cost, contrary to the previous algorithm. In all these results a significant amount of stringing can be seen. This is due to the way our slicer optimizes away filament retraction between support pillars. However, none of these thin plastic threads actually connect to the part, making cleaning very easy.

Figure 1 shows the *bunny peel* model after support removal, where two different materials were used for the part and the supports. Note how despite using very contrasted white/blue filaments, there are no significant white smears on the object surface outside of the downwards support anchors.

4.4. Robustness

We process a batch of 900 models extracted from the *Thingi10K* database [ZJ16] with voxel size 0.5mm. The algorithm successfully generated a support structure for every model in the dataset. An important future work would be to prove that the algorithm never encounters contradictions.

Our support elements are parallel to the x or y axes, making the structure dependent on the orientation of the part. Properly aligning the model can have a significant impact on the support size.

5. Conclusion

Our technique generates reliable support structures that avoid touching the part when possible in a reasonable time. The cost is independent of the object complexity and instead only scales with voxel grid size. In practice a solution is always found if it exists.

References

[DHL14] DUMAS, JÉRÉMIE, HERGEL, JEAN, and LEFEBVRE, SYLVAIN. "Bridging the Gap: Automated Steady Scaffoldings for 3D Printing". *ACM Transactions on Graphics* 33.4 (July 2014), 1–10. ISSN: 0730-0301, 1557-7368. DOI: 10/ghpm83 1, 3.

[Gum16] GUMIN, MAXIM. *Wave Function Collapse*. https://github.com/mxgmn/WaveFunctionCollapse. 2016 1, 2.

[INR13] INRIA. *IceSL Modeler and Slicer*. 2013 1.

[JXS18] JIANG, JINGCHAO, XU, XUN, and STRINGER, JONATHAN. "Support Structures for Additive Manufacturing: A Review". *Journal of Manufacturing and Materials Processing* 2.4 (Sept. 2018), 64. ISSN: 2504-4494. DOI: 10/gf6fcx 1.

[LEM*17] LIVESU, MARCO, ELLERO, STEFANO, MARTÍNEZ, JONÀS, et al. "From 3D Models to 3D Prints: An Overview of the Processing Pipeline". *Computer Graphics Forum* 36.2 (May 2017), 537–564. ISSN: 01677055. DOI: 10/gbmq9g 1.

[LGC*18] LIU, JIKAI, GAYNOR, ANDREW T., CHEN, SHIKUI, et al. "Current and Future Trends in Topology Optimization for Additive Manufacturing". *Structural and Multidisciplinary Optimization* 57.6 (June 2018), 2457–2483. ISSN: 1615-1488. DOI: 10/gdrwvj 1.

[Mer07] MERRELL, PAUL. "Example-Based Model Synthesis". *Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games - I3D '07*. The 2007 Symposium. Seattle, Washington: ACM Press, 2007, 105. ISBN: 978-1-59593-628-8. DOI: 10/cqqk94 1, 2.

[RRN19] RUSSELL, STUART JONATHAN, RUSSELL, STUART, and NORVIG, PETER. *Artificial Intelligence: A Modern Approach*. Pearson, July 2019. ISBN: 978-0-13-461099-3 1.

[SU14] SCHMIDT, RYAN and UMETANI, NOBUYUKI. "Branching Support Structures for 3D Printing". *ACM SIGGRAPH 2014 Studio*. SIGGRAPH '14. New York, NY, USA: Association for Computing Machinery, 2014. ISBN: 978-1-4503-2977-4. DOI: 10/ghtq3g 1.

[ZJ16] ZHOU, QINGNAN and JACOBSON, ALEC. "Thingi10K: A Dataset of 10,000 3D-Printing Models". *arXiv preprint arXiv:1605.04797* (2016). arXiv: 1605.04797 4.