Optimizing Evolutionary CSG Tree Extraction

Markus Friedrich
Institute for Computer Science LMU Munich
Munich, Germany
markus.friedrich@ifi.lmu.de

Pierre-Alain Fayolle
The University of Aizu
Aizu-Wakamatsu, Japan
fayolle@u-aizu.ac.jp

Thomas Gabor
Institute for Computer Science LMU Munich
Munich, Germany
thomas.gabor@ifi.lmu.de

Claudia Linnhoff-Popien
Institute for Computer Science LMU Munich
Munich, Germany
linnhoff@ifi.lmu.de

ABSTRACT

The extraction of 3D models represented by Constructive Solid Geometry (CSG) trees from point clouds is a common problem in reverse engineering pipelines as used by Computer Aided Design (CAD) tools. We propose three independent enhancements on state-of-the-art Genetic Algorithms (GAs) for CSG tree extraction: (1) A deterministic point cloud filtering mechanism that significantly reduces the computational effort of objective function evaluations without loss of geometric precision, (2) a graph-based partitioning scheme that divides the problem domain in smaller parts that can be solved separately and thus in parallel and (3) a 2-level improvement procedure that combines a recursive CSG tree redundancy removal technique with a local search heuristic, which significantly improves GA running times. We show in an extensive evaluation that our optimized GA-based approach provides faster running times and scales better with problem size compared to state-of-the-art GA-based approaches.

CCS CONCEPTS

• **Mathematics of computing → Combinatorial optimization**; *Graph algorithms*; **Combinatorial optimization**; • **Theory of computation → Evolutionary algorithms**; • **Computing methodologies → Shape representations**; **Reconstruction**; *Hierarchical representations*; *Shape modeling*;

KEYWORDS

3D Geometry Processing, CAD, CSG, 3D-Reconstruction, Evolutionary Algorithms

ACM Reference Format:
Markus Friedrich, Pierre-Alain Fayolle, Thomas Gabor, and Claudia Linnhoff-Popien. 2019. Optimizing Evolutionary CSG Tree Extraction. In *Genetic and Evolutionary Computation Conference (GECCO '19), July 13–17, 2019, Prague, Czech Republic,* Jennifer B. Sartor, Theo D'Hondt, and Wolfgang De Meuter (Eds.). ACM, New York, NY, USA, Article 4, 9 pages. https://doi.org/10.1145/3321707.3321771

GECCO '19, July 13–17, 2019, Prague, Czech Republic

ACM ISBN 978-1-4503-6111-8/19/07…$15.00
https://doi.org/10.1145/3321707.3321771

1 INTRODUCTION

Reverse engineering a 3D object from a set of points sampled on the surface of the object (also called a point cloud or point-set) is a common step in Computer Aided Design (CAD) pipelines. The surface points are typically obtained from laser scanners, RGB-D cameras, or techniques based on photogrammetry. In order to be able to manipulate or edit the recovered object, it is necessary to extract a higher-level representation of it. One expressive and intuitive approach for representing models in solid modeling is Constructive Solid Geometry (CSG), where complex rigid solids are defined by a tree structure consisting of Boolean set-operations (union, intersection, difference, complement) in the internal nodes and simple primitives (boxes, spheres, …) in the leaves.
Evolving CSG trees from a given set of primitives with the help of a Genetic Algorithm (GA)[1] such that the corresponding model represents the input point-set as accurately as possible is the topic of this paper. While the usage of Genetic Algorithms or Genetic Programming (GP) for this task has already been proposed in the past [5], computing times for sufficiently complex models (> 15 primitives) are known to be quite large (several hours). In this work, we propose a new GA-based approach that significantly outperforms existing GA-based approaches in terms of wall-clock times and scalability with problem size. In particular, this paper makes the following contributions:

- A point cloud filtering heuristic that reduces point cloud size, accelerating objective function evaluation.
- A problem partitioning scheme that accelerates the extraction process and simplifies the final merge of partial solutions.
- A 2-level improvement procedure applied to all individuals of a population that leads to faster convergence.
- A simplified objective function (compared to [5]) with a reduced set of model-independent, user-defined parameters that improves CSG tree shape controllability.

The rest of the paper is structured as follows: Basics on CSG trees are covered in Section 2. Related work on reverse engineering, point cloud to CSG conversion and the use of evolutionary methods in geometry processing is covered in Section 3. The description of our

[1] In this text, we use the term Genetic Algorithm, but since it operates on tree structures, one could have called it Genetic Programming instead.

approach is provided in Section 4, followed by several experiments and their results in Section 5. Section 6 concludes this paper.

2 BACKGROUND

First, we introduce some background material on how to represent CSG models, the primitives and operations that we support and how they are implemented. We also briefly describe the general point cloud to CSG model extraction pipeline, from which one step is handled by the approach described in this work.

2.1 CSG Tree Definition: Primitive Description and Boolean Set-Operations

CSG modeling is a technique that allows the iterative creation of complex geometric models by starting from simple shapes and combining them with Boolean set-operations. Thus, a CSG tree consists of simple primitives in the leaves combined with Boolean set-operations in the internal nodes.

Primitives. In this work, we represent primitives with implicit surfaces. In particular, we use signed distance functions whenever possible. For a solid S, its boundary surface ∂S is implicitly defined by the zero-set of its corresponding distance function f_S: $\{\mathrm{x} \in \mathbb{R}^3 : f_S(\mathrm{x}) = 0\}$. The surface normal at point $\mathrm{x} \in \mathbb{R}^3$ is given by the gradient of the distance function $\nabla f_S(x)$. We consider boxes, spheres, cones and cylinders as possible primitives.

Boolean Set-Operations. Currently supported Boolean set-operations are intersection, union, complement and subtraction. These operations are implemented using min- and max-functions [14]:

- Intersection: $S_1 \cap S_2 := \max(f_{S_1}, f_{S_2})$
- Union: $S_1 \cup S_2 := \min(f_{S_1}, f_{S_2})$
- Complement: $\overline{S} := -f_S$
- Subtraction: $S_1 \setminus S_2 := S_1 \cap \overline{S_2}$

where S_i is the solid corresponding to the set $\{\mathrm{x} \in \mathbb{R}^3 : f_{S_i} \leq 0\}$ $(i = 1, 2)$.

2.2 Extraction Pipeline: From Point Cloud to CSG Tree

The problem of extracting a CSG tree from a 3D point cloud relies on several steps:

(1) Point cloud pre-processing: Raw point cloud data is typically obtained from laser scanners, photogrammetry or RGB-D cameras, among others. The acquired data contains at least point coordinates and sometimes also comes with a surface normal at each point or even color information. If the point-wise surface normal is not available, it has to be estimated numerically, since it is used in the segmentation and fitting step as well as in the CSG extraction step. Additionally, data acquired from sensors is noisy in general and needs to be filtered, see [9] for a survey on point cloud filtering methods.
(2) Point cloud segmentation and fitting: In this step, the input point cloud is clustered in subsets, each corresponding to a possible type of primitive (box, sphere, ...). Parameters of the most probable primitive are fitted to the corresponding subset. Often, variants of RANSAC [15] are used in this step. Recently, methods leveraging large collections of data-sets used for training deep neural networks have started to appear in literature [12].
(3) CSG tree extraction and optimization: The final step is to extract a CSG tree that connects the fitted primitives of the previous step via Boolean set-operations. Additional optimization techniques can be applied during the CSG tree expression construction to simplify its structure. The approach introduced in this paper is concerned with handling this step.

3 RELATED WORK

While several approaches have been proposed over the years for reconstructing a 3D object from a set of points sampled on the surface, see for example [1] for a recent survey, we are only interested in approaches that allow the recovery of a CSG tree of the 3D object. A related problem is the conversion from a Boundary-Representation (B-Rep) model to a CSG model for which we describe existing solutions first.

3.1 B-Rep to CSG Conversion

Conversion from B-Rep to CSG consists in finding a CSG expression describing the same solid as the given B-Rep model. It was first investigated in 2D for polygons bounded by line segments, then extended for curved polygons in [17] and [16]. Optimization of the CSG expression and extension to 3D was discussed in [18] and [19], then later improved in [2]. Additionally, Shapiro and Vossler made the important remark in [19] that the boundary primitives obtained from the B-Rep model are not sufficient to describe the solid by CSG, and that additional separating primitives are necessary. The problem of these approaches is that they rely on exact representations, where patches form a partition of the input solid. In practice, input point clouds are often noisy, contain holes and thus lead to approximate representations that have negative impact on the achievable result quality.

3.2 Point Cloud to CSG Tree

To the best of our knowledge, the first work that dealt with extracting a CSG model from a point cloud was the work of Silva et al. in [21], where strongly typed Genetic Programming was combined with techniques to limit the size of the CSG tree. The results were limited to very simple point clouds, yet the generated CSG trees were quite complex. In [5], the problem of point cloud to CSG conversion was broken down into two stages: A segmentation/fitting procedure was performed in the first step. Fitted primitives were then combined in a second step by Genetic Programming to evolve a CSG tree. Results on more complicated point clouds were demonstrated and the extracted CSG trees have reasonable sizes.
Recently, the problem of extracting a CSG model from a point cloud has received a lot of attention [3, 20, 23]. In [20], a deep recurrent neural network is used to learn the production of CSG expressions. The results are limited to simple objects with small primitive-sets. In [23], primitives are clustered in groups of intersecting primitives. Each group contains only a few primitives and the CSG expression is generated by brute force. The final CSG expression is obtained by considering the union of all such CSG expressions. While providing good results for smaller models, scaling and precision issues arise

due to the need of a per-primitive sampling procedure for geometric quality evaluation. Du et al. use program synthesis [11] to evolve CSG expressions in [3]. To keep the time complexity low, they first apply a clustering algorithm to the input point cloud.

3.3 Evolutionary Based Approaches in Geometry Processing

Interfacing a geometric modeling kernel with Genetic Programming was proposed by Hamza and Saitou in [8] to automatically produce CSG models satisfying some constraints.
In [22], Evolutionary Algorithms are used to optimize CAD specification trees such that they satisfy structural optimization problems. Genetic Programming is used in [21] to produce a CSG tree from an input point cloud. Only simple point clouds are processed while generating complex CSG trees. The problem of point cloud to CSG tree conversion is split into two steps: Segmentation and fitting first, followed by Genetic Programming to combine the fitted primitives in a CSG tree in [5], allowing to handle more complex objects. The acceleration of the CSG tree extraction process is considered in [7] via geometric partitioning and parallelization of the computation on the different partitions. However, the proposed partitioning scheme is complex and does not guarantee tree size optimality after merge.

4 CONCEPT

We assume that the input point cloud is already segmented with a primitive fitted to each of the subsets, i.e. a primitive is assigned to each subset and its parameters are fitted. The input to our approach is thus a set of primitives P, where each primitive $p \in P$ is a 4-tuple (R, O, N, f_S) of geometric parameters R (e.g. center and radius of a sphere), surface points O, surface normals N and signed distance function f_S describing the primitive as a solid. The proposed extraction pipeline that outputs a CSG tree based on such input is made of different steps shown in Fig. 1 and described in the following subsections.

4.1 Intersection Graph Extraction

Our point cloud selection and problem partitioning approaches both rely on a graph called the intersection graph $I = (P, E)$ that has primitives P as vertices and edges E between vertices where corresponding primitives intersect. The graph is generated based on the primitive's parameters. For simple shapes based on Quadrics (e.g. cones, spheres, cuboids, cylinders, ...), a closed formula for intersection detection exists. If shapes get more complex, we approximate them with triangle meshes that serve as tight hulls for the shapes induced by the signed distance functions. In order to improve computational efficiency, we use an axis-aligned bounding box around the shape for early overlap testing.

4.2 Point Cloud Pre-Processing and Selection

Point cloud size has significant impact on the time needed to evaluate the objective function. Our method selects only six points for each connection between two primitives, reducing the computational complexity of an objective function evaluation to $O(|E|)$. The reduced point-set is still sufficient for model extraction.
In a first step, points of all primitives are pre-processed in order to cope with noisy input data. The following processing steps are applied:

Initial Filtering. We use the Median Absolute Deviation (MAD) measure $\text{mad}_i = \text{med}\left(|f_{S_i}(o_{ij}) - \text{med}(F_i)|\right)$ for the detection of outliers in primitive point clouds O_i, where F_i is the set of distances to the surface of primitive p_i for each point in O_i and $\text{med}(\cdot)$ is the median operator. A point o_{ij} is an outlier if it is more than three scaled MADs away from the median surface distance:

$$|f_{S_i}(o_{ij})| > 3c \cdot \text{mad}_i, \tag{1}$$

where the scale factor $c = -1/(\sqrt{2} \cdot \text{erfcinv}(3/2))$ with $\text{erfcinv}(\cdot)$ being the Inverse Complementary Error Function.

Surface Normal Alignment. Surface normals for each point are required to compute the matching quality between the point cloud and the CSG model. They are usually estimated and thus tend to be unstable at edges and regions with low point density. Since the primitive a point belongs to is known, we can evaluate the gradient of the primitive's distance function at that point to get a precise surface normal:

$$n_{ij} = \begin{cases} \nabla f_{S_i}(o_{ij}), & \text{if orientation of } p_i \text{ is outwards} \\ -\nabla f_{S_i}(o_{ij}), & \text{else} \end{cases}, \tag{2}$$

where $i \in [1, |P|]$ and $j \in [1, |O_i|]$. The surface orientation (outwards or inwards) is estimated by counting the primitive's original point normals that point outwards. If the result is greater than $|O_i|/2$, the primitive's orientation is considered to be outwards and inwards otherwise.

Point Projection. All points are relocated to their corresponding primitive's surface:

$$o_{ij} = o_{ij} - f_{S_i}(o_{ij}) \cdot \nabla f_{S_i}(o_{ij}) \tag{3}$$

This eliminates the effects of noise in point positions and further stabilizes the objective function evaluation.

Additional Filtering. We further increase robustness by filtering out points that are close (in terms of a user-defined ϵ) to neighboring primitives with a different surface orientation. In addition, points that are closer to neighboring primitives than to their assigned primitive are removed as well.

The second step involves the point selection mechanism, which works as follows: For each pair of primitives that share an edge in the intersection graph, the distance of each corresponding point to the centroid of the other primitive is computed and stored in a sorted list (thus there is one list per primitive). Then, the points with the maximum, the minimum and the median distance are extracted from that list, resulting in three points per primitive and a total of six points per primitive pair.

4.3 Partitioning

Before CSG extraction, the problem is partitioned based on the intersection graph. The partitioning scheme works as follows: Initially, primitives from I that are not relevant for partitioning are pruned. Then, so-called prime implicant primitives [17] are detected and disconnected from I. Finally, all connected components of the resulting graph with re-added pruned primitives are considered to

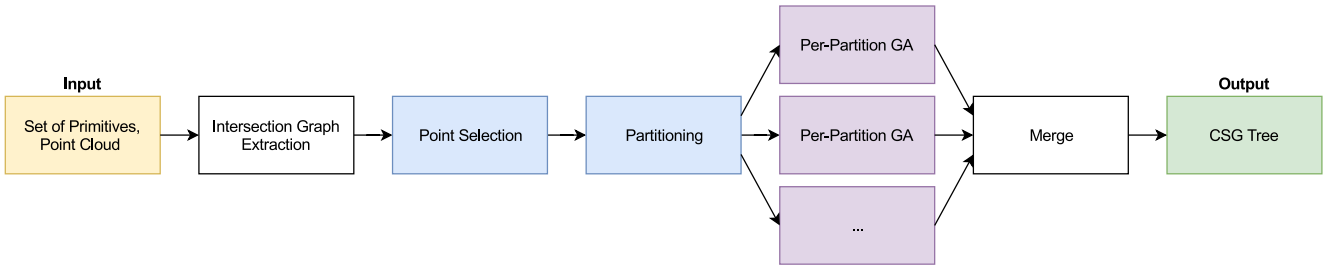
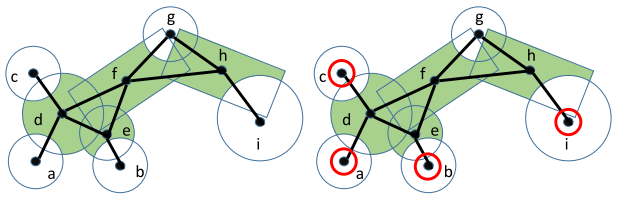


Figure 1: CSG tree extraction pipeline with input (orange), two optimization steps (blue), GA (purple) and output (green).

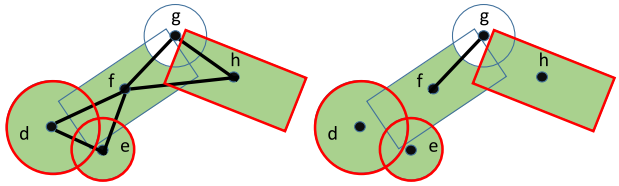
be valid partitions of I. Our proposed scheme has the following advantages:

- Computational complexity of the problem is significantly reduced.
- Per-partition solutions can be computed in parallel.
- Probability to converge to a global minimum is increased for smaller problem instances.
- Primitives in a partition are likely to be spatially close which results in CSG trees that are more intuitively editable.

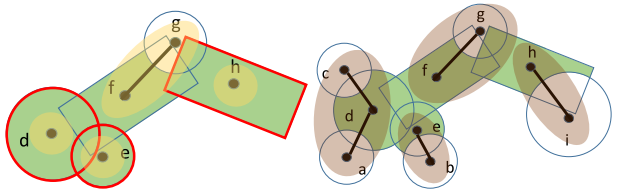
The different steps of the partitioning scheme are explained in the following paragraphs and exemplified in Fig 2.



(a) Primitives with intersection graph. (b) Primitives to be pruned (red).



(c) Prime implicants (red) (d) Disconnected prime implicants.



(e) Connected components (yellow) (f) Partitions (brown).

Figure 2: Partitioning steps. Surface to cover in green.

Pruning. All primitives (vertices) that only have a single edge are pruned from the intersection graph. In the example depicted in Fig. 2(b), this is the case for primitives c, a, b and i.

Detecting Prime Implicants. Prime implicants are detected by checking for each primitive $p_i \in P$ if no points of neighbor primitives are inside p_i and normal vectors of points of p_i have the same orientation as surface normals of p_i. See Fig. 3 for an illustration. Results of this step are shown in Fig. 2(c).

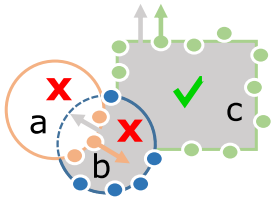


Figure 3: Primitive c is a prime implicant since there is no point corresponding to neighbor primitives that is located inside of c and none of the normal vectors of c's points have a different orientation than the surface normals of c.

Disconnecting Prime Implicants. Vertices of all found prime implicants are disconnected by removing all corresponding edges from the intersection graph (see Fig. 2(d)).

Identifying Connected Components. The connected components of I build up the basis for the partitions (see Fig. 2(e)). Finding all connected components in an undirected graph like I has a computational complexity of $O(|P| + |E|)$ and can be implemented with a breadth-first search.

Reattaching Pruned Primitives. Now that a partitioning of I exists, the last step is to reattach vertices (primitives) that have been pruned in the first step of the process. This results in the final partitions (see Fig. 2(f)).

4.4 Per-Partition GA

The proposed GA extracts per-partition CSG trees by solving a multi-objective optimization problem. It uses a similar mutation and crossover operator and Tournament Selection for parent selection as the GA proposed in [5]. See Fig. 4 for an overview of the whole process with newly developed components in green. Please note that for partitions with less than 3 primitives, the GA is replaced with a brute force approach that tries all possible combinations.

Ranking. The fitness function that needs to be maximized has two parts: A geometry term ($E_{geo}(\cdot)$) that measures how close a CSG tree matches the point cloud and a tree size/depth penalty

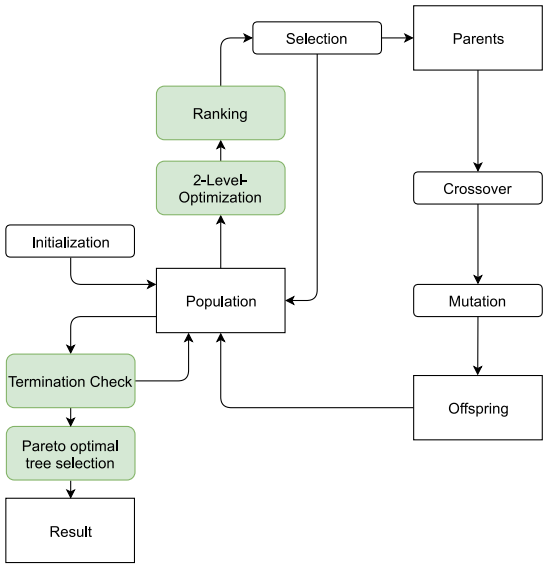


Figure 4: The GA used for extracting per-partition CSG trees (adapted from [4]). Proposed extensions are highlighted in green.

term that restricts the tree size and depth. The complete functional reads

$$E(t) = E_{geo}(t) - \alpha \cdot \left\{ \beta \cdot \frac{\mathrm{s}(t)}{\mathrm{s}_{max}} + (1-\beta) \cdot \frac{\mathrm{d}(t)}{\mathrm{d}_{max}} \right\}, \tag{4}$$

where t is the tree candidate, $s(\cdot)$ returns the size of t, $d(\cdot)$ its depth. s_{max} and d_{max} are maximum size and depth of the trees in the current population. The parameter α controls the influence of the size penalty term and β weights tree size against tree depth, enabling fine-grained control of the tree's size-to-depth ratio. $E_{geo}(\cdot)$ evaluates the geometrical fitness of t returning a value in the range $[0, 1]$ and reads

$$E_{geo}(t) = \frac{1}{|P||O_i|} \sum_{i=1}^{|P|} \sum_{j=1}^{|O_i|} \begin{cases} 1, & \text{if } |f_t(o_{ij})| < \epsilon \wedge \nabla f_t(o_{ij}) \cdot n_{ij} \geq 0 \\ 0, & \text{else} \end{cases}, \tag{5}$$

where f_t is the semialgebraic signed distance function corresponding to the CSG tree t. The geometric term checks for each point if it is (a) on the surface induced by t and (b) if its normal has the same orientation as that surface. The ranking implementation makes use of a CSG tree caching mechanism. Experiments have shown that approximately 50% of objective function evaluations can be avoided that way.

2-Level Local Improvement. All CSG tree candidates are subjected to a 2-level improvement mechanism before ranking. This can be compared to a very narrow local search that is meant to impose certain invariants on the candidates that help their overall fitness. In contrast to general Memetic Algorithms [10, 13], e.g., the local search uses a domain-specific deterministically generated neighborhood around the respective tree candidate at stake. The introduction of this local improvement mechanism is one of the contributions of this paper.

The first step recursively traverses a tree and searches for redundant structures, like:

- Operations with identical primitives as operands
- Operations with the null primitive as one or more operands.

The null primitive acts as a placeholder with which redundant structures are replaced. The procedure is repeated until the particular tree does not contain redundant structures anymore.
The second step traverses a tree and searches for operations with both operands being primitives. For each operator found, it checks if the operands are connected in the intersection graph. If not, a randomly chosen operand is replaced with a neighboring primitive. Then, all possible operators (union, intersection, difference) are tested and the one with the best geometry score (see Equation 5) replaces the current one.

Termination Check. The proposed termination criterion exploits the fact that the geometry term of the objective function evaluates to 1 for trees that perfectly match the point cloud. As long as the population does not contain any tree t with $E_{geo}(t) = 1$, the GA continues (or ends after a maximum iteration count has been reached). If a tree in the population has a perfect geometry score, an iteration counter is started. After a certain amount of additional iterations (we empirically found out that $8 \cdot |E|$ is a sufficient upper bound), the GA is terminated.
The idea behind the stop criterion is that once a tree that perfectly matches the input point cloud has been found, the GA should try to find a minimal tree in terms of size and depth.
In [5], the geometry term is not normalized hence its maximum is not known in advance. Thus, the GA is terminated if a user-controlled maximum number of iterations has been reached, which makes it harder to aim for a geometrically perfect representation.

Pareto Optimal Tree Selection. In each GA iteration, we search and store the pareto optimal tree of all trees and populations ranked so far. After termination, this tree is taken as the resulting tree.
In [5], best tree selection is solely based on its objective function value. This might result in the selection of geometrically suboptimal results due to the size penalty included in the objective function.

4.5 Merging

Due to the merge-friendly nature of the employed partitioning scheme, the final CSG tree is just the union of all per-partition trees. The merging step is simple and does not worsen the size-optimality of the final CSG tree obtained after merging since it does not depend on a correct detection of similar subtree structures like [7] does.

5 EVALUATION

We evaluated our CSG tree recovery approach in different scenarios (with and without partitioning, with and without local improvement, ...) on eight point clouds that represent both common and edge cases. See Fig. 11 for renderings of the used models and Fig. 5 for a resulting CSG tree for model $M0$. Each scenario was evaluated three times in order to get an adequate understanding of the result variance for identically parametrized runs. The details of used data-sets are depicted in Table 1. GA parameters that were used throughout the experiments are summarized in Table 2. All

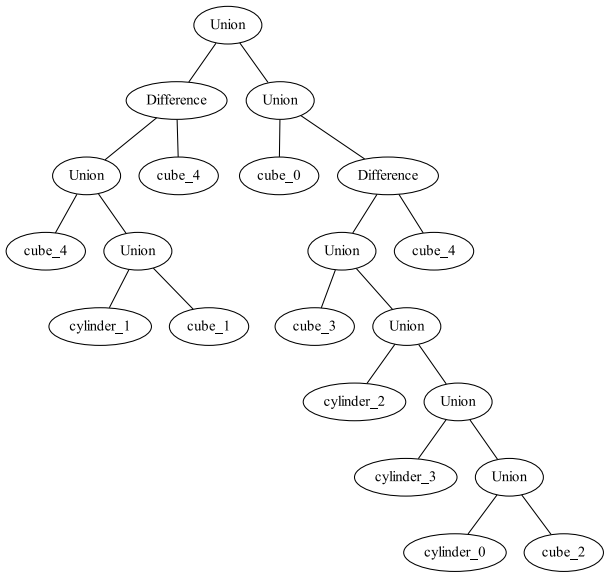


Figure 5: Resulting CSG tree for model $M0$.

	M0	**M1**	**M2**	**M3**
# Primitives	9	5	7	4
# Points	25k	18.9k	26.3k	25.9k
# Partitions	1	1	1	1
	M4	**M5**	**M6**	**M7**
# Primitives	38	40	37	86
# Points	10.8k	105.1k	111.7k	190.5k
# Partitions	1	16	13	8

Table 1: Details for the different input point clouds used in the evaluation. "# Points" corresponds to the total number of points in the input point clouds. "# Primitives" corresponds to the total number of primitives used for the description of the object. "# Partitions" corresponds to the number of partitions computed by the partitioning scheme (see Section 4.3).

Parameter Name	**Value**
Population size	150
# Best parents	2
Crossover probability	0.4
Mutation probability	0.3
Subtree replacement probability	0.5
Tournament selection parameter	2
Tree size weight α	1.0
Tree size/depth weight β	0.7

Table 2: GA parameters used for all conducted experiments.

experiments were conducted on a virtual machine with 8 GB of RAM and 32 assigned 2.2GHz Xeon cores.

5.1 2-Level Local Improvement

We evaluated our approach on all data-sets in different improvement scenarios: (1) No improvement, (2) using both improvement steps, (3) using only the second improvement step. Refer to Section 4.4 for a description of both steps. For this experiment, all runs were stopped if the current population contained a tree with $E_{geo}(t_{\text{best}}) = 1.0$. Furthermore, the partitioning scheme was disabled such that the GA is applied to the full problem instance instead of running multiple GAs in parallel, one for each detected partition. The results (number of iterations and time taken until $E_{geo}(t_{\text{best}}) = 1.0$) are illustrated in Fig. 6 and Fig. 7. We can observe that using an improvement strategy is always preferable to not using any (both in terms of number of iterations and running times). The only exception is object $M5$, where applying the complete 2-level improvement strategy gives worse results than using none at all. Interestingly, we can also observe that on the simpler models ($M0$, $M1$, $M2$, $M3$) the 2-level improvement strategy almost always provides better results than using the second improvement step only (exception: $M3$). However, on the more complex objects ($M4$, $M5$, $M6$, $M7$), using only the second improvement step is preferable to using both steps in most cases (exception: $M4$).

This seems to suggest that the first improvement step might not be beneficial for the extraction of complex objects. This effect can be explained by investigating the evolution of geometry score and best tree size over executed iterations as shown in Fig. 8 for model $M5$: With the first improvement step enabled, redundant structures are removed too aggressively which results in significantly smaller tree sizes. This hinders the GA to keep a sufficient level of diversity in the population. While showing a higher convergence rate in the beginning, the 2-level improvement strategy results in a slow convergence mostly in the missing last 1%. Using only the second improvement step outperforms the baseline (without improvement) since redundancies are preserved, which keeps population diversity on a sufficiently high level (visible in the larger tree sizes compared to full 2-level improvement). Additionally, the best primitive selection performed by the second improvement step leads to a higher convergence rate while its computational effort is neglectable.

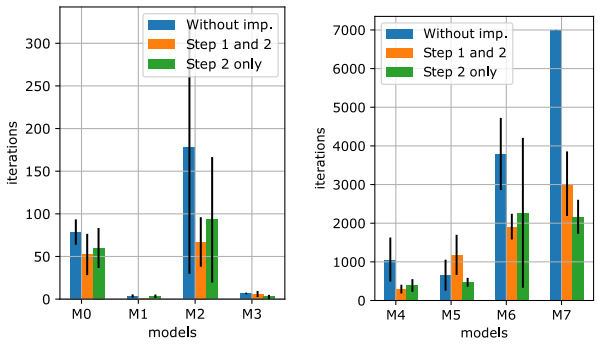


Figure 6: Number of iterations needed to reach $E_{geo}(t_{\text{best}}) = 1.0$.

5.2 Point Selection

We measured the running times with and without enabled point selection for all data-sets together with second step only improvement. For better comparability, only the first 200 iterations of each

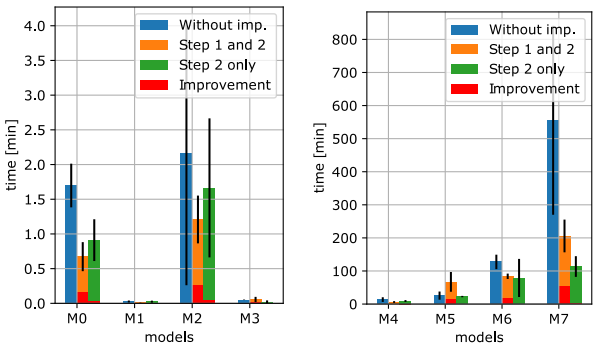


Figure 7: Time needed to reach $E_{geo}(t_{\mathbf{best}}) = 1.0$. Red bars indicate the time spent in the improvement steps.

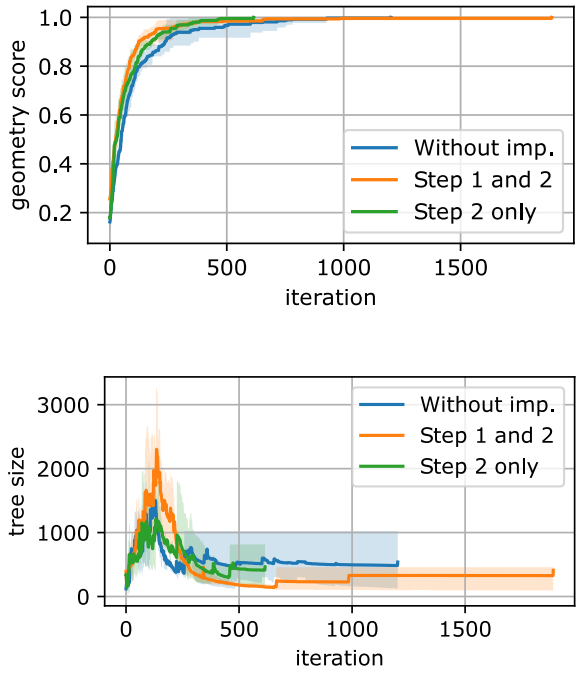


Figure 8: Geometry scores and tree sizes for model $M5$ and all iterations until $E_{geo}(t_{\mathbf{best}}) = 1.0$. Standard deviation in semi-transparent colors.

run were considered (for models $M1$ and $M3$, 75 and 66 iterations are considered since this is the maximum number of iterations reached in our experiments). Fig. 9 shows the results, whereas Table 3 depicts the sizes of the selected point-sets. It is clearly visible that point selection improves performance on the tested models significantly by at least a factor of 1.34 (model $M4$). As expected, the models with the largest corresponding point- and primitive-sets ($M5, M6, M7$) profit the most from point selection (factors: 19.39, 16.26 and 19.07). $M4$ profits the least, which is mainly due to the comparably low ratio between original point-set size and selected point-set size, which is only 58.06. Together with the impact of parallel tree ranking that is enabled throughout all benchmarks, this results in a rather insignificant positive effect of point selection in this particular case.

	M0	**M1**	**M2**	**M3**
# Selected Points	96	48	114	36
Reduction Ratio	260.42	393.75	230.7	719.44
	M4	**M5**	**M6**	**M7**
# Selected Points	186	432	564	780
Reduction Ratio	58.06	243.29	198.05	244.23

Table 3: Number of selected points for all models and the reduction ratio $\frac{\#\text{Points}}{\#\text{Selected Points}}$.

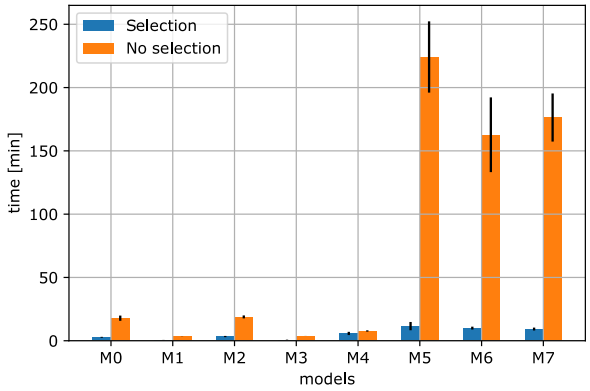


Figure 9: Running times for all models for the first 200 iterations with and without point selection (for models $M1$ and $M3$, 75 and 66 iterations were considered).

5.3 Partitioning

The effects of partitioning were evaluated using all models with more than a single partition ($M5$, $M6$, $M7$). These results are obtained with both improvement steps enabled as well as with only the second improvement step enabled. In addition, we also compare single-threaded per-partition GA execution with a multi-threaded variant that spawns a thread for each partition. Please note that wall-clock times for the partitioning itself (connection graph computation, pruning and prime implicant detection) are not considered since they are in all cases negligible.

The results are presented in Fig. 10. The positive effect of partitioning on computation times is significant and ranges from a speed-up factor of 3.46 ($M7$, second improvement step only, single-threaded) to 32.85 ($M5$, second improvement step only, per-partition parallelism). The inter-model difference in speed-up factors is due to the different numbers of partitions (8 for $M7$, 16 for $M5$) and model complexities ($M7$:86 primitives, $M5$:40 primitives). Another advantage of the partitioning approach lies in the reduced running time variance most visible for model $M6$.

An important aspect is the comparison of the two different improvement modes. The results provided in Section 5.1 indicate that applying the full 2-level improvement strategy is beneficial for most of the smaller models. Since partitioning leads to smaller partial models, it is expected that it performs better than second step only improvement. This is true for model $M6$ and $M7$ but not for $M5$. As part of our future work, we plan to investigate the influence of the model topology (as indicated by the intersection graph) on the performance of the two considered improvement modes.

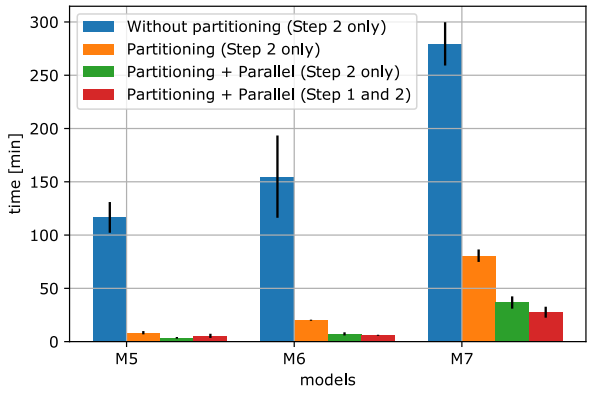


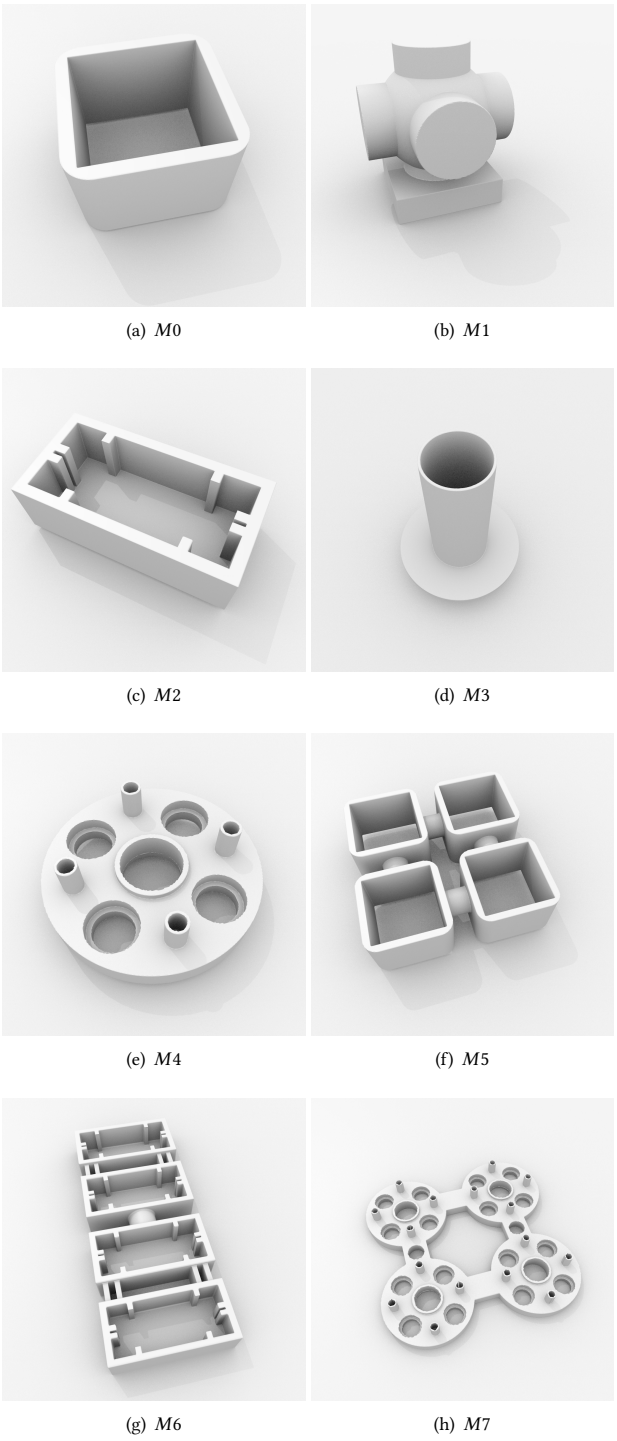
Figure 10: Running times for all models $M5$, $M6$ and $M7$ with and without partitioning for different improvement modes.

6 CONCLUSION & FUTURE WORK

We presented a new GA-based CSG tree extraction scheme that comprises of three major optimization mechanisms: A point selection technique that significantly reduces objective function evaluation effort, a merge-friendly partitioning scheme that divides the problem into smaller, simpler to solve problems and a 2-level improvement approach applied to all trees of a population in each iteration which further improves running times.

For future work, we would like to investigate the potential of structural pattern detection. The idea is that the CSG tree for similar structures should be extracted only once. In addition, a primitive detection and fitting pipeline could increase the usability of our system significantly.

Another interesting research direction yet to be explored is the use of alternative metaheuristics for solving the CSG tree extraction and optimization problem. Of particular interest for us are methods that rely at least partially on formulations suitable for the execution on Quantum Annealing (QA) hardware as proposed in [6]. An interesting research question is to what extent the problem can be transformed into a suitable formulation and what parts of it remain to be solved by classic algorithms. We believe that the potential of such hybrid quantum algorithms for our problem domain is worth exploring more thoroughly.



(a) $M0$ (b) $M1$ (c) $M2$ (d) $M3$ (e) $M4$ (f) $M5$ (g) $M6$ (h) $M7$

Figure 11: Renderings of all models considered in the evaluation. For rendering, triangle meshes were sampled from CSG trees that were extracted using the proposed pipeline.

REFERENCES

[1] Matthew Berger, Andrea Tagliasacchi, Lee M Seversky, Pierre Alliez, Gael Guennebaud, Joshua A Levine, Andrei Sharf, and Claudio T Silva. 2017. A survey of surface reconstruction from point clouds. *Computer Graphics Forum* 36, 1 (2017), 301–329.

[2] Suzanne F Buchele and Richard H Crawford. 2004. Three-dimensional halfspace constructive solid geometry tree construction from implicit boundary representations. *Computer-Aided Design* 36, 11 (2004), 1063–1073.

[3] Tao Du, Jeevana Priya Inala, Yewen Pu, Andrew Spielberg, Adriana Schulz, Daniela Rus, Armando Solar-Lezama, and Wojciech Matusik. 2018. InverseCSG: Automatic Conversion of 3D Models to CSG Trees. In *SIGGRAPH Asia 2018 Technical Papers (SIGGRAPH Asia '18)*. ACM, New York, NY, USA, Article 213, 16 pages. https://doi.org/10.1145/3272127.3275006

[4] Agoston E Eiben, James E Smith, et al. 2003. *Introduction to evolutionary computing*. Vol. 53. Springer.

[5] Pierre-Alain Fayolle and Alexander Pasko. 2016. An evolutionary approach to the extraction of object construction trees from 3D point clouds. *Computer-Aided Design* 74 (2016), 1–17.

[6] Sebastian Feld, Markus Friedrich, and Claudia Linnhoff-Popien. 2018. Optimizing Geometry Compression using Quantum Annealing. In *Accepted at the IEEE Workshop on Quantum Communications and Information Technology 2018 (IEEE QCIT 2018)*. 1–6.

[7] Markus Friedrich, Sebastian Feld, Thomy Phan, and Pierre-Alain Fayolle. 2018. Accelerating Evolutionary Construction Tree Extraction via Graph Partitioning. In *Proceedings of WSCG International Conference on Computer Graphics, Visualization and Computer Vision*.

[8] Karim Hamza and Kazuhiro Saitou. 2004. Optimization of Constructive Solid Geometry Via a Tree-Based Multi-objective Genetic Algorithm. In *Proceedings of GECCO*. 981 – 992.

[9] Xian-Feng Han, Jesse S. Jin, Ming-Jie Wang, Wei Jiang, Lei Gao, and Liping Xiao. 2017. A review of algorithms for filtering the 3D point cloud. *Signal Processing: Image Communication* 57 (2017), 103 – 112. https://doi.org/10.1016/j.image.2017.05.009

[10] Natalio Krasnogor and Jim Smith. 2005. A tutorial for competent memetic algorithms: model, taxonomy and design issues. *IEEE Transactions on Evolutionary Computation* 9, 5 (2005), 474–488.

[11] A Solar Lezama. 2008. *Program synthesis by sketching*. Ph.D. Dissertation. U. C. Berkeley.

[12] Lingxiao Li, Minhyuk Sung, Anastasia Dubrovina, Li Yi, and Leonidas Guibas. 2018. Supervised Fitting of Geometric Primitives to 3D Point Clouds. *arXiv preprint arXiv:1811.08988* (2018).

[13] Nasimul Noman and Hitoshi Iba. 2008. Accelerating differential evolution using an adaptive local search. *IEEE Transactions on evolutionary Computation* 12, 1 (2008), 107–125.

[14] A. Ricci. 1973. A constructive geometry for computer graphics. *Comput. J.* 16, 2 (1973), 157–160.

[15] Ruwen Schnabel, Roland Wahl, and Reinhard Klein. 2007. Efficient RANSAC for point-cloud shape detection. *Computer graphics forum* 26, 2 (2007), 214–226.

[16] Vadim Shapiro. 2001. A convex deficiency tree algorithm for curved polygons. *International Journal of Computational Geometry & Applications* 11, 02 (2001), 215–238.

[17] Vadim Shapiro and Donald L Vossler. 1991. Construction and optimization of CSG representations. *Computer-Aided Design* 23, 1 (1991), 4–20.

[18] Vadim Shapiro and Donald L Vossler. 1991. Efficient CSG representations of two-dimensional solids. *Journal of Mechanical Design* 113, 3 (1991), 292–305.

[19] Vadim Shapiro and Donald L Vossler. 1993. Separation for boundary to CSG conversion. *ACM Transactions on Graphics (TOG)* 12, 1 (1993), 35–55.

[20] Gopal Sharma, Rishabh Goyal, Difan Liu, Evangelos Kalogerakis, and Subhransu Maji. 2018. CSGNet: Neural Shape Parser for Constructive Solid Geometry. (2018).

[21] Sara Silva, Pierre-Alain Fayolle, Johann Vincent, Guillaume Pauron, Christophe Rosenberger, and Christian Toinard. 2005. Evolutionary computation approaches for shape modelling and fitting. In *Progress in Artificial Intelligence*. Springer Berlin Heidelberg, 144–155.

[22] Daniel Weiss. 2009. *Geometry-based structural optimization on CAD specification trees*. Ph.D. Dissertation. ETH Zurich.

[23] Qiaoyun Wu, Kai Xu, and Jun Wang. 2018. Constructing 3D CSG Models from 3D Raw Point Clouds. *Computer Graphics Forum* 37, 5 (2018), 221–232.