Assembly Sequence Planning for Complex real-world CAD data

Dissertation
zur Erlangung des Grades
"Doktor der Naturwissenschaften"

am Fachbereich Physik, Mathematik und Informatik
der Johannes Gutenberg-Universität in Mainz,

vorgelegt von
Sebastian Dorn, M. Sc.
geboren in Wertheim

Mainz, den 09. Mai 2022

Berichterstatter:
Prof. Dr. Elmar Schömer
Johannes Gutenberg-Universität, Mainz

Prof. Dr. Ernst Althaus
Johannes Gutenberg-Universität, Mainz

Tag der mündlichen Prüfung: 14. Juli 2022

D77

ABSTRACT

This work investigates algorithms for digital assembly sequence planning. The focus here is on finding feasible assembly sequences for a real-world data set from the automotive industry. Flexible fastening elements that have to be deformed when being disassembled, a large workspace in which the components are located and a very high number of assembled parts push conventional assembly sequence algorithms to their technical and temporal limits when it comes to real-world data. In three consecutive chapters of this work, we investigate how to push these boundaries into the feasible range, meaning that we are conducting pioneering research into the calculation of feasible assembly sequences for real-world data.

The first part of this thesis presents an approximation of a *general Voronoi diagram* which subdivides the workspace at the maximum clearance. We combine a common voxel propagation algorithm with a novel memory-saving data structure. We then discuss different ways to extract a *general Voronoi diagram graph*. This graph is a roadmap which provides a roughly estimated translational disassembly path for assembled parts, called *Voronoi path*. The second part of this work introduces our new path planner *Expansive Voronoi Tree (EVT)*. For a given part, the EVT searches along the according Voronoi path for a collision-free disassembly path, consisting of arbitrary translations and rotations. Our experiments show that the EVT finds shorter paths more reliably within shorter calculation time than conventional path planning algorithms. In the third part, we present our framework for finding feasible assembly sequences. We are the first to calculate feasible assembly sequences for a real-world data set. By using two new algorithms, we detect parts that are still enclosed, which significantly reduces the number of path planning requests and, in turn, calculation time. Our framework divides the disassembly process for an assembled part into a NEAR- and FAR planning phase. Using the path planner from Hegewald *et al.* (2022) which is designed for deformable fasteners, we unlock parts in the NEAR range and then, in the FAR planning phase, navigate them to a nearby goal position using our EVT. The disassembly paths found are then presented in our new, informative and easy-to-read *Assembly Priority Graph*. Compared to other representation methods our

graph is compact but also shows a high number of feasible assembly sequences. This graph is the result of the interplay of all previously mentioned findings and thus the final result of this work.

ZUSAMMENFASSUNG

In dieser Arbeit werden Algorithmen zur virtuellen Montageplanung untersucht. Der Fokus liegt hierbei auf dem Finden von zulässigen Montagereihenfolgen für einen Realdatensatz aus der Automobilbranche. Flexible Befestigungselemente, welche während des Ausbauprozesses deformiert werden müssen, ein großer Arbeitsraum, in dem sich die Bauteile befinden sowie die sehr große Anzahl von Montageteilen bringen die gängigen Montagesequenz-Algorithmen bei Realdaten an ihre technischen und zeitlichen Grenzen. In dieser Arbeit werden diese Grenzen in drei aufeinander aufbauenden Kapiteln zum Machbaren hin verschoben, so dass wir als Erste zulässige Montagesequenzen für Realdaten berechnen können.

Im ersten Teil der Arbeit wird die Approximation eines *general Voronoi diagrams* vorgestellt, welches den Arbeitsraum an den Stellen mit maximalem Freiraum unterteilt. Der Voxel-propagierende Algorithmus arbeitet auf unserer neuartigen, Arbeitsspeicher sparenden Datenstruktur. Anschließend diskutieren wir verschiedene Möglichkeiten einen *general Voronoi diagram graph* zu extrahieren. Dieser Graph wird als Straßennetz interpretiert, welches einen grob geschätzten, translatorischen Ausbaupfad (sogenannter *Voronoi Pfad*) für montierte Bauteile liefert. Der zweite Teil der Arbeit stellt unseren neuartigen Pfadplaner *Expansive Voronoi Tree (EVT)* vor. Dieser sucht für ein Bauteil anhand seines Voronoi Pfades nach einem kollisionsfreien Ausbaupfad, welcher aus beliebigen Translationen und Rotationen besteht. Unsere Experimente zeigen, dass der EVT schneller und zuverlässiger kürzere Pfade findet als gängige Pfadplanungsalgorithmen. Im dritten Teil stellen wir unser Framework zum Finden von zulässigen Montagesequenzen vor. Wir sind die Ersten, die für einen Realdatensatz zulässige Montagesequenzen berechnen können. Wir erkennen mit zwei neuartigen Algorithmen aktuell noch eingeschlossene Bauteile, verringern somit die Anzahl von Pfadplanungsanfragen und folglich die Berechnungsdauer enorm. Unser Framework unterteilt den Demontagevorgang von einem montierten Bauteil in einen Nah- und Fernbereich. Wir lösen mit dem für deformierbare Befestigungselemente ausgelegten Pfadplaner von Hegewald *u. a.* (2022) Bauteile im Nahbereich und navigieren diese anschließend mittels unserem EVT im Fernbereich zu einer nahegelegenen Zielposition.

Die gefundenen Ausbaupfade werden anschließend in unserem neuartigen, aussagekräftigen und leicht zu lesenden *Assembly Priority Graph* präsentiert. Verglichen mit gängigen Darstellungsmethoden ist unser Graph kompakt, beinhaltet aber dennoch eine Vielzahl von unterschiedlichen Montagesequenzen. Dieser Graph ist das Resultat vom Zusammenspiel aller erwähnten Ergebnisse und somit das finale Ergebnis dieser Arbeit.

DANKSAGUNG

An dieser Stelle möchte ich mich herzlich bei meinem Betreuer/Doktorvater als auch in gleichem Maße bei meiner Betreuerin bedanken. Sie beide haben mir die Erstellung der Dissertation ermöglicht und mit ihren wertvollen Ratschlägen, Ideen und konstruktiver Kritik maßgeblich zum Gelingen dieser Arbeit beigetragen. Ebenfalls möchte ich mich bei dem Zweitgutachter bedanken. Dank geht auch an meine Doktoranden-Kollegen, die mich in Diskussionen mit ihrem Fachwissen unterstützt haben und die Zeit im Allgemeinen kurzweiliger gestaltet haben. Darüber hinaus bedanke ich mich bei meinem Betreuer und meinem Teamleiter bei der Mercedes-Benz AG für das mir entgegengebrachte Vertrauen und die mir gegebenen Freiheiten.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1

INTRODUCTION

The production process describes all the tasks and processes that are needed to produce a technical product. This process consists of many highly complex sub-processes. As shown in Figure 1, these include the fabrication of each individual part, the *assembly process* of these parts and the final quality check. This thesis focuses on the assembly process. This process describes the complete procedure of turning a set of individual parts into one product - i.e. when should which part be assembled, and how? This gives rise to the following questions, for example: Should a part be assembled by a robot or a mechanic? What force should be applied in order to tighten a specific screw? How many assembly lines are needed? Can two parts be assembled in parallel? Which tools (e.g. hammers, wrenches, etc.) are needed? Are the ergonomic requirements for the mechanics met? These detailed questions should ideally be answered with a view to ensuring that the assembly process is as fast, cost-effective and ergonomic as possible.

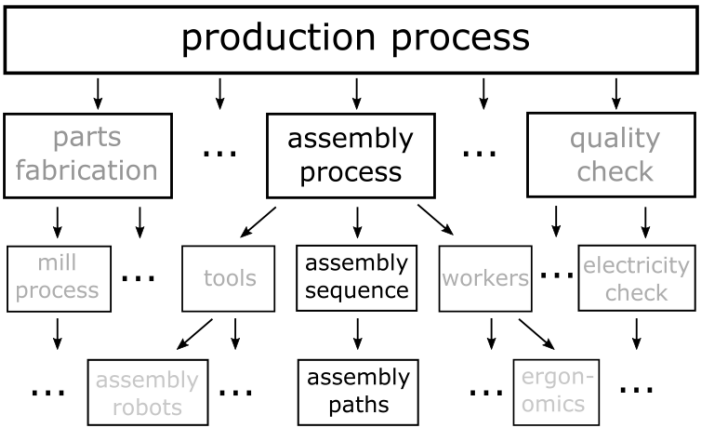

Fig. 1: Production and some of the sub-processes.

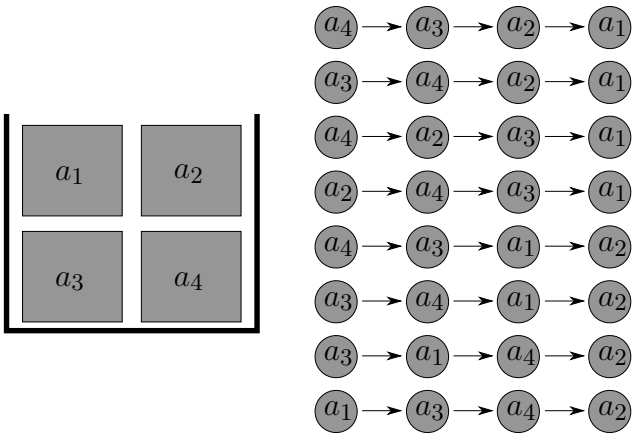

Fig. 2: Left: A two-dimensional assembly with parts a_1, a_2, a_3, a_4. Right: All feasible assembly sequences.

An essential part of the assembly process is the *assembly sequence* which describes the order in which each part is to be assembled. A possible assembly sequence for the two-dimensional example shown in Figure 2 is $a_4 \to a_3 \to a_2 \to a_1$. However, this simple example comprising only four parts has as many as eight different sequences. The assembly sequence $a_1 \to a_3 \to a_4 \to a_2$ is much more complicated since a_3 has to be navigated after a_1 is in its final position. Passing this narrow passage takes more time and requires more concentration on the part of the mechanic. In addition, a mechanic may need special tools to navigate a_3 around the corner at the end.

The procedure for determining assembly sequences, is as follows: First, check which parts are not yet assembled. Second, assemble one of these parts if possible; then go back to the first step. The procedure ends when all parts are assembled. To determine whether a part can be assembled, an *assembly path* needs to be found. A geometrically feasible assembly path navigates a part from its start position to a goal position. The term "feasible" in this context means that a part does not collide with other parts. Consequently, finding such a path is crucial for finding assembly sequences. The process of calculating feasible paths is part of the research field of *motion planning*.

In summary, a well-chosen assembly sequence with appropriate assembly paths is crucial for the overarching objective - namely, the assembly process is ergonomic and cost-effective.

A major challenge occurs in the aforementioned second step, in which a part is added to the assembly. An assembled part might blocks subsequent parts. Therefore, if a_1 and a_2 are assembled, for example, it is not possible to assemble a_3 and a_4. The

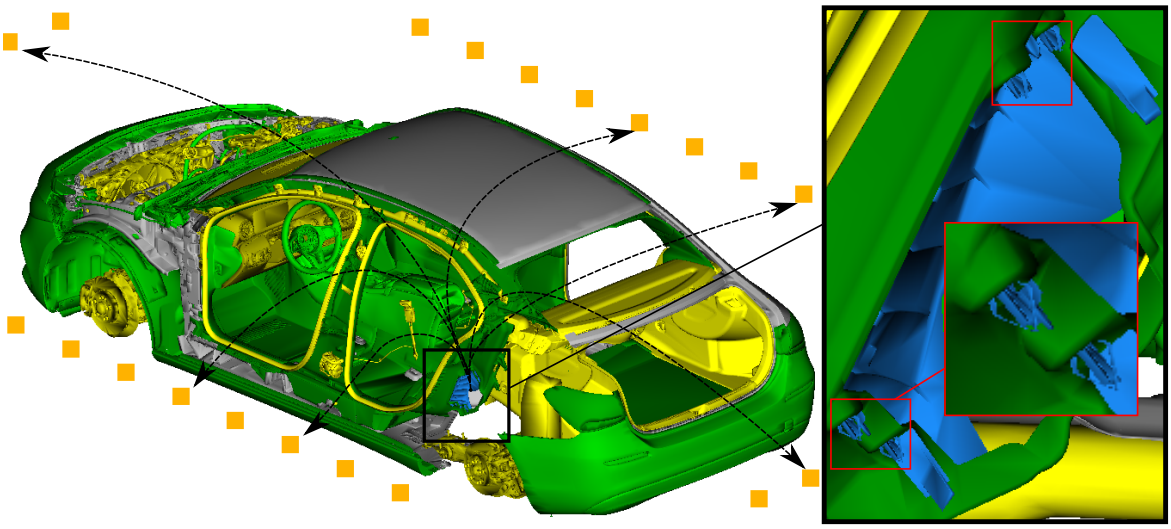

Fig. 3: The data represents a subset of the Mercedes-Benz A-Class. The surrounding orange squares are the goal positions for the assembled parts. The dashed arrows show possible feasible disassembly paths for the blue-highlighted covering part. This covering part is fastened with the red-marked flexible clips shown on the right.

conventional *assembly-by-disassembly* technique solves this problem. It starts with the assembled state and iteratively disassembles one part after another. To perform the assembly sequence, the found *disassembly sequence* and *disassembly paths* are reversed. Therefore, we change the point of view from assembly to disassembly. Furthermore, a disassembly sequence is also used for repairing or replacing a broken part. Here, a disassembly (sub-)sequence for the parts that need to be removed first is needed. The fewer parts need to be disassembled, the faster repairs can be performed.

In addition to major challenge of finding a disassembly sequence that yields the most cost-effective assembly process, however, each disassembly sequence must fulfill the basic property of geometric feasibility. The term ”geometrically feasible” in this context means that each part can be disassembled along the corresponding path without colliding with components that are still assembled. The research field *assembly sequence planning (ASP)* addresses how to find a geometrically feasible disassembly sequence. In general, the number of possible disassembly sequences increases exponentially with the number of assembled parts. It is therefore not possible to search for all disassembly sequences, even for small examples. The minimum requirement is to find at least one disassembly sequence. Nevertheless, for this sequence to be used in practice, it and the associated paths should be meaningful and realistic so that the sequence found can be used in an assembly process that is ergonomically and economically satisfactory, if not optimal.

Conventional ASP frameworks proceed as follows: A main loop iterates over every still-assembled part and makes a rigid body motion planning attempt until each part is successfully disassembled. This procedure is long known and well-studied for academic examples (Mello and Sanderson, 1988), (Hadj *et al.*, 2018), (Ebinger *et al.*, 2018).

However, real-world data - as shown in Figure 3 - presents a number of unresolved problems for the research field of ASP. No framework currently exists that is capable of performing ASP for complex real-world data. We categorize these problems into the following four points (*i*) - (*iv*). In addition, we derive from these problems a range of requirements for an ASP framework.

Flexible fastening elements (*i*): In addition to screws and nuts, clips in all their varieties are among the most common fastening elements. As shown in the zoomed image section in Figure 3, clips are modeled in a relaxed state. This means that in order to find a feasible disassembly path, the clips need to be deformed. These clips will otherwise inevitably cause collisions, which previous algorithms are generally incapable of handling.

Requirement (*i*): Addresses the problem of flexible fastening elements during motion planning.

Large workspace (*ii*): A complex data set such as the vehicle shown has a large workspace. This large workspace provides a multitude of possibilities for navigating a part to a goal position. Each dashed arrow shows a possible disassembly path for the blue-highlighted part. However, they differ significantly in terms of length and practical manageability. For example, navigating the blue highlighted part through the windshield is much more complex than simply navigating it through the rear-left door.

Requirement (*ii*): Find appropriate disassembly paths.

Numerous assembled parts (*iii*): The data set shown consists of more than 800 parts. The aforementioned main loop from ASP frameworks generally requires, depending on the number of parts, a quadratic number of motion planning attempts. A complicated motion planning attempt like this can take up to a few minutes. This makes the calculation times for large data sets such as this very long and therefore impracticable.

Requirement (*iii*): Speed up the ASP framework; specifically reduce the number of motion planning attempts.

Table 1: The requirements for the NEAR- and FAR planning phases.

planning phase	flexible parts	optimized path
NEAR	✓	x
FAR	x	✓

Difficult presentation of the assembly sequences (*iv*): As mentioned above, the number of disassembly sequences increases exponentially depending on the number of components. Even if only a subset is calculated, the quantity of disassembly sequences is difficult to represent. Therefore, a representation is required that is much more compact than the one shown in Figure 2.

Requirement (*iv*): Find a compact and meaningful representation for the found disassembly sequences and disassembly paths.

Our contribution: Our main contribution with this thesis is an ASP framework that fulfills requirements (*i*) - (*iv*). To achieve this, we present new scientific contributions to the requirements (*ii*) - (*iv*). To solve (*i*) in the context of ASP, we use an existing work and integrate it in our framework.

We will now analyze problems (*i*) - (*iv*) and outline our main approaches to solving these problems.

(*i*) + (*ii*): Without taking into account the deformation of flexible fastening elements it is not possible to unlock them. Once unlocked, however, the clips do not need to undergo any further deformation. The slight deformation of the clips is relevant only near the special counterpart to which the clip is attached. This deformation is negligible for navigation in large workspaces. Assembled parts have so little clearance in their immediate environment that the disassembly paths near the installed position do not vary to any great extent, i.e. they are all very similar. Once the part is unlocked, however, there are a number of ways that i can be navigated to a goal position (as shown in Figure 3). We can conclude, therefore, that different requirements apply to different disassembly phases. We call the unlocking phase the *NEAR planning phase* and subsequent navigation in the large workspace the *FAR planning phase*. The subdivision of the motion planning process was first developed by Masan (2015). The planning phases are disjoint and have different requirements which are summarized in Table 1. This is why our framework uses a different motion planner for each planning phase.

(*i*): For the NEAR planning phase, we use the *Iterative Mesh Modification Planner (IMMP)* from Hegewald *et al.* (2022). In a preprocessing step with a geometry-based heuristic, this motion planner detects flexible fastening elements and shrinks them. It then uses a sampling-based strategy to search for a feasible disassembly path NEAR the installed position.

(*ii*): Next, we describe our thoughts and observations regarding the FAR planning phase. Finding optimized paths is a well-known and long-standing problem in the research field of motion planning. Motion planning in the context of ASP, however, has a special characteristic: The workspace is the same for all assembled parts at a certain point in time. In addition, the workspace changes only slightly when a part has been removed. As an example, two neighboring screws can use the same disassembly path in the FAR planning phase. It therefore makes sense to determine possible disassembly paths that can be used by all parts as a basis.

For this purpose, we subdivide the entire workspace using the *general Voronoi diagram (GVD)* (see Figure 4, left), which belongs to the research field of computational geometry. The GVD subdivides the workspace at the points that have maximal clearance with respect to their surroundings. From the GVD, we extract a motion planning roadmap, the *general Voronoi diagram graph (GVDG)*. Since it is promising to disassemble a part along a path with the maximum clearance with respect to its surroundings, we use the GVDG for the FAR planning phase as a rough estimate of the translational part of a disassembly path. However, approximating the GVD for complex real-world data is a challenging task. A common approach for achieving this is to voxelize the scene. These voxels then propagate iteratively to their neighbors. If two voxels with different starting voxels meet, a GVD voxel is created. However, the underlying data structure needed for this propagation process is a three-dimensional grid. The cubic memory needed for this grid exceeds the existing memory for fine voxelizations. We present our novel data structure *Voronoi voxel history*, which stores only the current wavefront and not the entire three-dimensional grid. Memory usage, therefore, is roughly quadratic rather than cubic, which we demonstrate in our experiments.

After a successful NEAR planning phase, the FAR planning phase takes over. We search within the GVDG for a short estimated translational disassembly path, a *Voronoi path (VP)*. Figure 4 right, shows VPs for assembled parts to a goal position. Finally, using our novel *Expanisve Voronoi Tree (EVT)* motion planner, we search along the VP for a feasible disassembly path. It is possible that the disassembly path that is to be found along a VP contains narrow passages. Our EVT is therefore based on

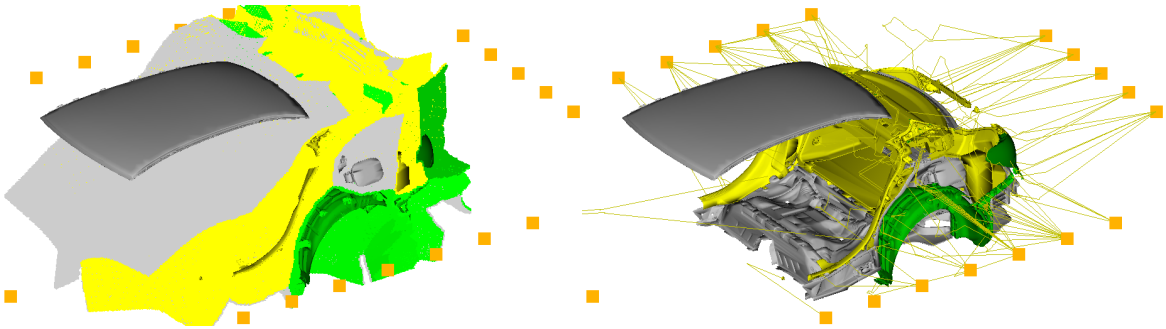

Fig. 4: The data set is a subset from Figure 3. Left: The corresponding general Voronoi diagram. Right: Extracted Voronoi paths along the GVD.

the 'Expansive Spaces Tree (EST)' (Hsu *et al.*, 1997) which is optimized for narrow passages. The disadvantage of the EST is that it explores the space only very slowly. We can overcome this problem by pushing our EVT sampling in the forward direction of the VP. In this way, our EVT inherits the capability to overcome narrow passages from the EST and is speeded up by sampling the translational part along the VP.

In summary, the combination of the IMMP from Hegewald *et al.* (2022) for the NEAR planning phase and our novel EVT for the FAR planning phase results in an overall motion planning process that fulfills requirements (*i*) and (*ii*).

(*iii*): The motion planning attempts require almost the entire computation time of an ASP framework. Reducing the number of motion planning attempts is crucial for speeding up an ASP framework. We developed two different algorithms that reduce the number of motion planning attempts. The first algorithm is based on the GVDG. The GVDG does not provide a VP for each assembled part. We analyze the reasons for this and conclude that if the GVDG cannot provide a VP for an assembled part, the part is enclosed by its surroundings, meaning that no feasible disassembly path for the part currently exists. As a result, no motion planning attempts are performed for parts that do not have a VP. Our second algorithm stores all parts with the part to be removed during the motion planning attempt. We now assume that our motion planners in the NEAR- and FAR planning phase find a path, if one exists. If a part cannot be disassembled, a motion planning attempt must be made to disassemble the part only when one of the parts in collision already has been removed. Our experiments show that the combination of these two algorithms speeds up our framework by a factor of 3. This is how we fulfill requirement (*iii*).

(*iv*): A number of ways to represent disassembly sequences have been identified, e.g. AND/OR-graphs, Petri-nets or Diamond graphs (Michniewicz, 2019). The

aforementioned representations need storage capacity that is linear with respect to the number of disassembly sequences. However, the number of disassembly sequences is generally exponential with respect to the number of parts. These representations are therefore not useful for a data set containing hundreds of parts. We extend the common blocking graph (Wilson, 1992) and create the *assembly priority graph (APG)*. For all parts and disassembly paths, the APG represents the parts which need to be removed first. The representation is easy to read and does not require significant storage capacity. Thus, the APG fulfills the last requirement (*iv*).

We combine all the aforementioned algorithms in our own ASP framework. We want to emphasize that our benchmark data set is an almost complete car. This car has 761 parts which consist of over 6 million triangles. This data set is therefore much more complex and comprehensive than the data used so far in the literature. Our experiments for the data set from Figure 3 show that these algorithms are capable of handling complex data. Finally, we show that we are the first to calculate and represent the disassembly sequences identified for a complex, real-world scenario from the automotive industry. For this, we use our APG, which is meaningful, easy to read and extendable.

In summary, this thesis offers new contributions to the research fields of computational geometry (Dorn *et al.*, 2020), motion planning (Dorn *et al.*, 2021) and assembly sequence planning (Dorn *et al.*, 2022). Each one of our novel algorithms operates in isolation, but their full potential is unleashed when deployed in combination in the context of our ASP framework.

1.1 Structure

Our work is subdivided into six chapters. Our main aim is to enable ASP for real-world data. Following this introduction, therefore, we discuss the special requirements and properties for real-world CAD data in the context of ASP. In the third chapter, we introduce the GVD and propose our propagation algorithm with its novel RAM-saving data structure. The chapter also contains a detailed discussion of the different ways to create the GVDG. The fourth chapter introduces the motion planning problem and explains why we subdivide the motion planning process into the NEAR- and FAR planning phases. We present our novel EVT motion planner for the FAR planning phase. The fifth chapter looks at ASP. We first introduce our novel assembly priority

graph, after which we present our assembly sequence planning framework, the first one capable of handling a real-world CAD data set from the automotive industry. Our framework makes extensive use of the results from the two previous chapters. Finally, we summarize our findings and outline a range of proposals concerning future works in the field of assembly sequence planning. Despite the interdependence between Chapters 3, 4 and 5, all three chapters can be read as standalone contributions.

2

TEST DATA AND BACKGROUND

2.1 Test Data

The main contribution of this dissertation is assembly sequence planning for real-world CAD data. Compared with examples from academia, real-world scenarios - such as our data from the automotive industry - come with a range of special requirements and challenges. The main differences between real-world data and academic data sets are the high number of parts that are positioned in a large workspace, multiple goal points and flexible fastening elements. We present these in detail below.

2.1.1 Data Size

We evaluate our algorithms based on two subsets of a representative real-world data set which contains a high number of complex parts in different positions. A graphical representation can be found in Figure 5. The attributes of the data are listed in Table 2. Column "#parts" gives the number of parts comprising the body shell and all assembled parts, whereby each part is represented as a three-dimensional triangulated mesh. "#triangles" shows the number of triangles in the assembly as a whole (bodyshell and assembled parts). The dimensions of the oriented bounding box containing the assembly are shown in "OBB size". With 761 parts comprising more than 12 million triangles, data set b) contains the most parts of a complete vehicle. We excluded the doors, bonnet and tailgate because these parts need to be closed and opened many times during the assembly process, which significantly complicates the assembly process. The wheels are not included either because they are not part of the standard CAD

Table 2: The attributes of the data sets.

Test Data	#parts		#triangles	OBB size [mm]		
data set	*body shell*	*assembled parts*	*assembly*	x	y	z
a)	50	137	885,000	2,623	1,786	1,292
b)	100	661	6,135,050	4,617	1,991	1,317

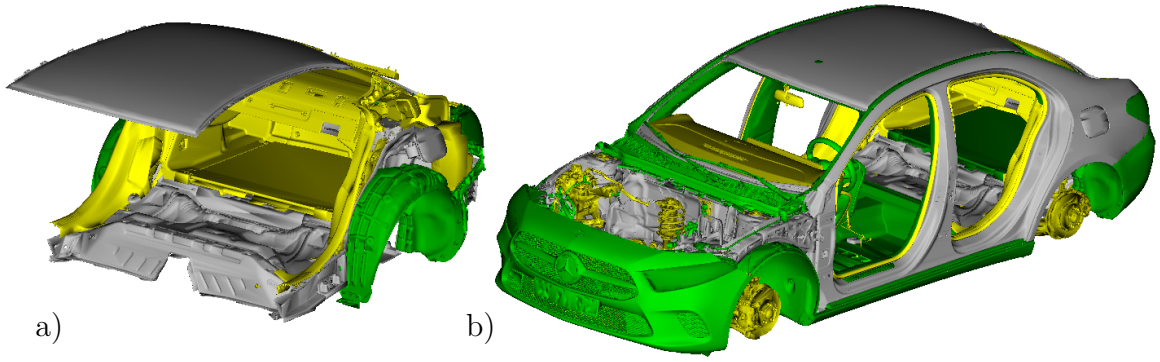

Fig. 5: The test data represent two different subsets of the Mercedes-Benz A-Class. The grey sites belong to the body shell and the different-colored ones are the assembled sites.

data set. The engine block is confidential and so not included either. Table 3 shows a classification of the assembled parts. The major parts are 'fasteners', followed by the 'normal parts' and the minor parts are 'others'. The boundaries between the classes are fluid, of course, but this table provides an overview of the rough composition of the parts. To provide more information about the data set, Figure 6 shows two fasteners and four normal parts with the number of triangles that represent the appropriate part. Some parts classified as 'others' are shown in the separate Section 2.1.3.

Data set a) is a subset of b) and is used to show the scalability of our algorithms. In addition, the slimmed-down version a) is less densely packed and some algorithms can

Table 3: A classification of the 661 assembled parts from data set b).

class	description	count	
		#	%
'fasteners'	screws, nuts, bolts, clips	396	60
'normal parts'	steel sheets, covering plates, wheels, seats	165	25
'others'	wires, foamed material, rubber seal	100	15

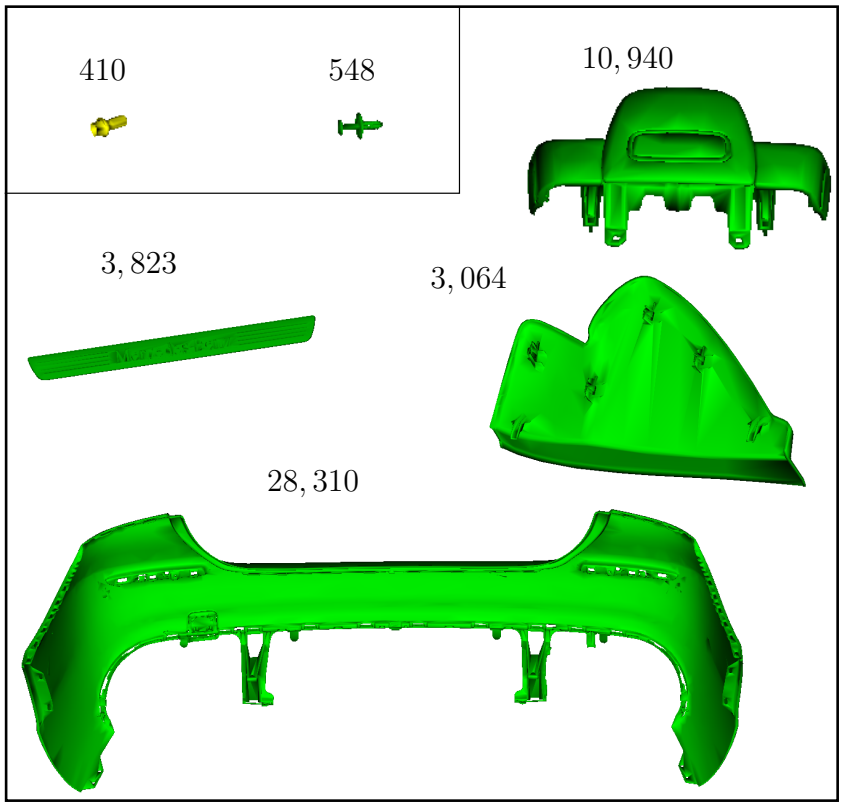

Fig. 6: Two 'fasteners' in the top-left corner and four 'normal parts'. The number next to the parts indicates the number of triangles. The parts are not shown to scale.

be better presented visually. Both data sets have significantly more parts than the data sets used in comparable works for ASP (Aguinaga *et al.*, 2008), (Ebinger *et al.*, 2018), (Ou and Xui, 2013), (Yu and Wang, 2013), (Jiménez, 2013), (Hadj *et al.*, 2018). These data sets include, for example, different kinds of puzzles (< 20 parts), a drill (22 parts), toy plane (32 parts), engine (57 parts), petrol engine (18 parts), and die cutter (29 parts). In our scenario, the body shell can be considered as a fixed part on which a set of parts are assembled. For the rest of this work, we define the assembly as follows Definition 1.

Definition 1 (*Assembly*)

An assembly $\bar{A} := (O, A)$ *consists of the body shell* O *and the assembled parts* $A := \{a_1, ..., a_n\}$. *The body shell and each assembled part are represented as a triangulated mesh.*

Definition 2 (*Mesh*)

A three-dimensional triangulated mesh $m := (V, T)$ *is defined with its finite set of nodes* $V \subset \mathbb{R}^3$ *and the finite set of triples of indices* $T := \{(t_1^1, t_1^2, t_1^3), ..., (t_m^1, t_m^2, t_m^3) | 1 \leq t_i^j \leq$

$|V|\}$ *which indicates with the indices of* V *the set of triangles. Another representation of* m *can be done with* $m := \{\triangle_1, ..., \triangle_n\}$ *where each triangle is defined as* $\triangle := \{v_i, v_j, v_k | v_i, v_j, v_k \in \mathbb{R}^3\}$.
In this work, we define that a point p_m *is an element of* m *if it lies on the surface of* m*:* $p_m \in m \Leftrightarrow \exists \triangle \in m \; p_m \in \triangle$.

Collision checks:

We perform the collision check for two meshes using the *indexed bounding volume hierarchy (IBVH)* (Gottschalk *et al.*, 1996). The IBVH of a mesh is a tree and is created as follows: The root is an *oriented bounding box (OBB)* which contains the complete geometry. This OBB is subdivided into two smaller OBBs which contain the corresponding sub-geometry. This procedure is performed iteratively until an OBB contains only a few triangles. The IBVH is calculated once when the mesh is loaded.

Two meshes are now checked for collision using their IBVH as follows: A check is first performed to determine whether, two root OBBs are in collision. If, the two children octrees are then checked pairwise for collision. Perform these steps iteratively until the leaves are reached and then check the triangles for collision.

2.1.2 Goal Points

Finding a disassembly path for assembled parts is one of the major challenges in ASP. An assembled part starts in an initial position and is disassembled if a certain condition is fulfilled. In rigid motion planning, this condition is usually if the robot's and obstacles' OBBs are disjoint or if the robot has reached a specific goal point. In real-world scenarios, not every OBBs disjoining feasible position is useful, e.g. a goal position on top of a car is unfavorable for the assembly process. In our work, therefore, we define a set of standardized goal positions, so called *body-in-white points (BIWP)*, around our car (see Figure 7). The BIWP is a set of pre-defined positions from which a worker could potentially start an assembly process. A disassembly process is therefore successful if the assembled part to be disassembled reaches one of the *BIWP*.

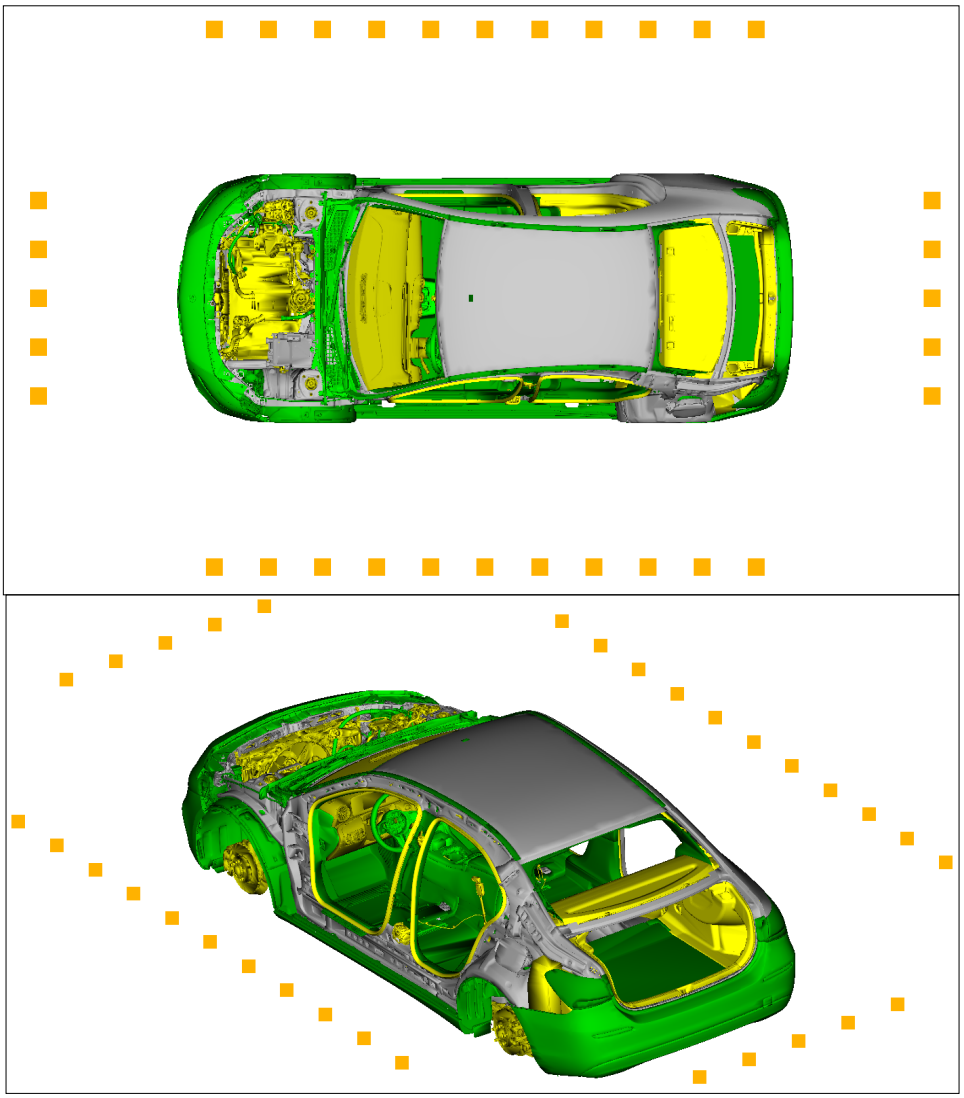

Fig. 7: The orange dots surrounding the car represent the body-in-white points.

2.1.3 Flexible Parts and Initial Collisions

Real-world scenarios involve numerous fastening parts. Screws and nuts are the most common fastening elements. Due to mesh-simplification threads are not modeled in detail. The thread of a screw is therefore modeled as a cylinder and the internal thread of a nut as a normal hole. A cylinder that is too big is therefore stuck in a hole that is too small which results in initial collisions.

In addition to screws and nuts, clips are also common fastening elements. Clips come in various shapes and forms (see Figure 8), but what all clips have in common is that they

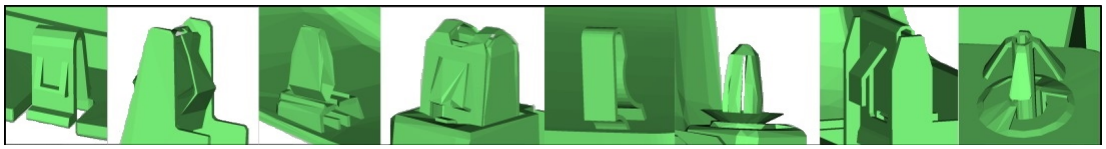

Fig. 8: Eight different types of clips. All of them needs to be deformed during the disassembly process.

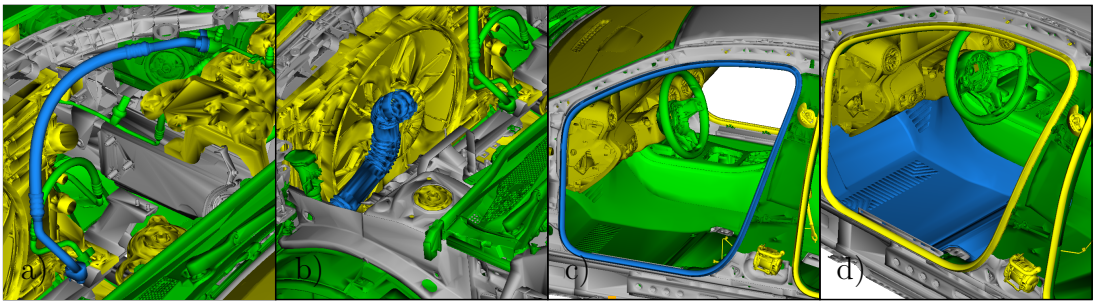

Fig. 9: Four flexible parts that are part of our test data: a) cable, b) hose, c) sealing, d) fabric covering.

need to be deformed during the disassembly process. In addition, clips are modeled in the relaxed state, which can also cause initial collisions.

Clips contain flexible subparts which need to undergo 'minor' deformation in order for a disassembly path to be found. Some parts, however, are 'completely' deformable, e.g. cables, hoses, sealings and fabric coverings. See Figure 9 for some examples of completely flexible parts. All these parts are characterized by their very high deformability, a feature that is extremely challenging and time-consuming to simulate.

2.1.4 Test Setup

All results and experiments are benchmarked on a laptop with an Intel i7-6820HQ (2.70GHz) processor, 32 GB of RAM and an NVIDIA Quadro M2000M graphics card. The software (64bit binary) is written in C++ and compiled with MS Visual C++ 15.9. All algorithms use floating precision and are executed - with the exception of certain renderingsteps - on a single core of the CPU.

3

GENERAL VORONOI DIAGRAM (GRAPH)

This chapter focuses on the approximation of the *general Voronoi diagram (GVD)*, the extracting of the *general Voronoi diagram graph (GVDG)* and how we use it for assembly sequence planning.

We start by introducing the well-known ordinary Voronoi diagram (VD), along with the medial axis (MA) and describe how we generalize the VD to define the GVD. We present a voxel-based propagation algorithm that approximates the GVD. This algorithm can work with two different underlying data structures: (*i*) a complete three-dimensional distance field: (*ii*) our *novel* hash-table-based data structure, the so called *Voronoi voxel history (VVH)*. The VVH requires roughly only a bit more than quadratic memory usage and the complete distance field cubic memory usage with respect to grid resolution. However, this reduced RAM usage comes with the cost of a longer running time. The experiments on our test data quantify the calculation time and memory storage required for both data structures.

Finally, we introduce the *general Voronoi diagram graph (GVDG)*. The GVDG is a graph-based roadmap for subsequent motion planning, which is covered in Chapter 4. The analysis of the structure shows that the GVDG can contain *isolated sites*. An isolated site is an assembled part for which the GVDG does not provide a connection to a BIWP. The occurrence of an isolated site, however, arises depends on how the GVDG is generated. We present and discuss three different approaches for generating the GVDG. Our novel *outside-to-inside (OTI)* propagation algorithm ensures that isolated sites occur only if no feasible disassembly path exists. This is useful for when consider a later ASP framework (see Chapter 5) because it allows assembled parts to be detected that cannot currently be removed. This significantly reduces the number of unsuccessful motion planning attempts and. in turn, the calculation time.

3.1 General Voronoi Diagram (GVD)

The ordinary Voronoi diagram (VD) is an important geometric data structure with numerous applications (Okabe *et al.*, 2000) such as illustrating the influence area of schools, hospitals or shops, i.e. the Voronoi region indicates the households which are nearest to the respective school building. For a given set of sites $S \subset \mathbb{R}^2$, the VD partitions the plane into Voronoi regions VR where every point in a region has the same nearest input site $s \in S$. In the ordinary case, sites are points and the metric used is the Euclidean distance. An example of an ordinary VD is shown in Figure 10. The VD is defined as follows (Descartes, 1644):

Definition 3 (*Voronoi diagram*)

Given $S \subset \mathbb{R}^2$ a finite set of points.
The Voronoi region VR of a site $s \in S$ is defined as $VR(s, S) := \{p \in \mathbb{R}^2 | d(s, p) < d(s', p)\ \forall s' \in S \setminus \{s\}\}$ where $VR(S) := \cup_{s \in S} VR(s, S)$ defines the union of all Voronoi regions.
The Voronoi edge VE of two sites $s_1, s_2 \in S$ is defined as $VE(s_1, s_2, S) := \{p \in \mathbb{R}^2 | d(s_1, p) = d(s_2, p) \wedge d(s_1, p) < d(s', p)\ \forall s' \in S \setminus \{s_1, s_2\}\}$.
The Voronoi node VN of three sites $s_1, s_2, s_3 \in S$ is defined as $VN(s_1, s_2, s_3, S) := \{p \in \mathbb{R}^2 | d(s_1, p) = d(s_2, p) = d(s_3, p) \wedge d(s_1, p) \leq d(s', p)\ \forall s' \in S \setminus \{s_1, s_2, s_3\}\}$.
The VD is then given as $VD(S) := \mathbb{R}^2 \setminus VR(S)$.

The VD is also the union of all Voronoi edges and Voronoi nodes.

The ordinary VD can be generalized in many different ways. For example, arbitrary input sites S which are placed in higher dimensional spaces than simply points in the plane, can be considered. In addition, the underlying distance metric can be arbitrary. We generalize the VD in the way that we consider three-dimensional meshes as input sites. We therefore need to transfer the above definition. In the three-dimensional space, three sites are needed for defining a Voronoi edge and four sites for defining a Voronoi node. Two sites define a *Voronoi face*. In addition, we store the *medial axis (MA)* (see Figure 11 for a two-dimensional example), also called *skeleton*, in our GVD. The MA is basically the (G)VD of a part itself. In a three-dimensional scenario, it allows us to navigate through parts with holes. The GVD is therefore defined as follows:

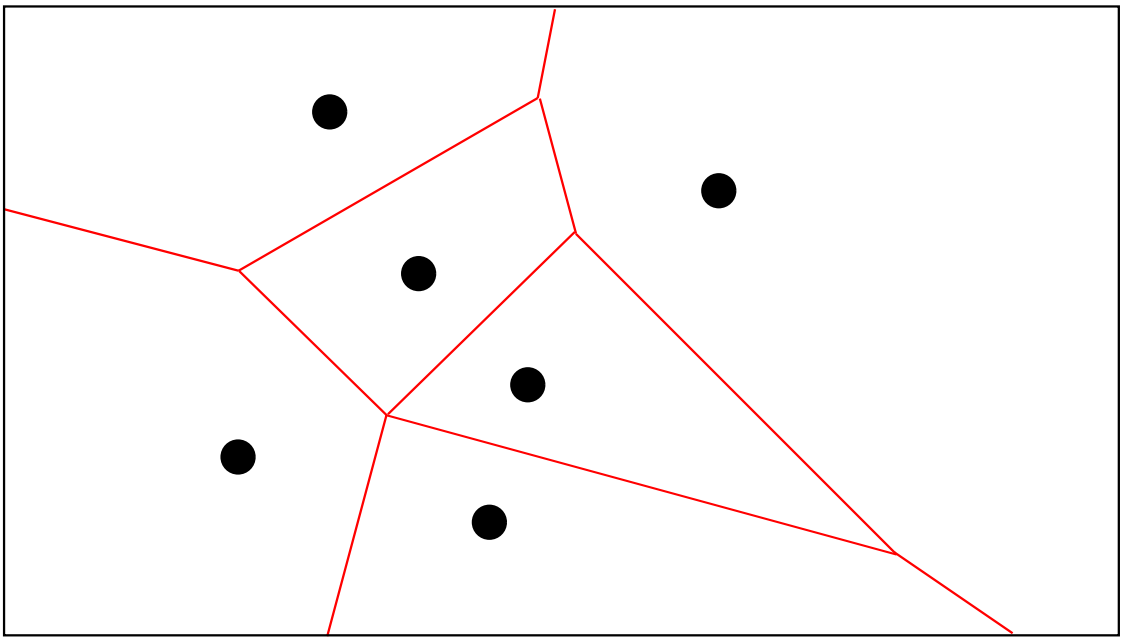

Fig. 10: The black dots represent the input sites S, the white areas the Voronoi regions and the red lines the Voronoi diagram.

Definition 4 (*General Voronoi diagram*)

Given the input data $S := \{s_1, ..., s_n\}$ *which consists of* n *three-dimensional meshes. We define the distance* d *from a point* $p \in \mathbb{R}^3$ *to a mesh* s *as* $d(p, s) := \min_{p_s \in s} \|p_s - p\|_2$.

The Voronoi region VR *of a site* $s \in S$ *is defined as* $VR(s, S) := \{p \in \mathbb{R}^3 | d(s, p) < d(s', p)\ \forall s' \in S \setminus \{s\}\}$ *where* $VR(S) := \cup_{s \in S} VR(s, S)$ *defines the union of all Voronoi regions.*

The Voronoi cell VC *of a site* $s \in S$ *is defined as the boundary of its Voronoi region.*

We broaden our GVD by the medial axis $MA(S) := \cup_{s \in S} MA(s, S)$. *The medial axis* MA *of a site* $s \in S$ *is defined as* $MA(s, S) := \{p \in \mathbb{R}^3 | (\exists p_s, q_s \in s, p_s \neq q_s, d(s, p) = d(p_s, p) = d(q_s, p) \wedge d(p_s, q_s) \geq \delta_{MA} \wedge (d(s, p) < d(s', p)\ \forall s' \in S \setminus \{s\})\}$ *with* $\delta_{MA} \geq 0$.

The Voronoi face VF *of sites* $s_1, s_2 \in S$ *is defined as* $VF(s_1, s_2, S) := \{p \in \mathbb{R}^3 | d(s_1, p) = d(s_2, p) \wedge d(s_1, p) < d(s', p)\ \forall s' \in S \setminus \{s_1, s_2\}\}$.

The Voronoi edge VE *of sites* $s_1, s_2, s_3 \in S$ *is defined as* $VE(s_1, s_2, s_3, S) := \{p \in \mathbb{R}^3 | d(s_1, p) = d(s_2, p) = d(s_3, p) \wedge d(s_1, p) < d(s', p)\ \forall s' \in S \setminus \{s_1, s_2, s_3\}\}$.

The Voronoi node VN *of sites* $s_1, s_2, s_3, s_4 \in S$ *is defined as* $VN(s_1, s_2, s_3, s_4, S) := \{p \in \mathbb{R}^3 | d(s_1, p) = d(s_2, p) = d(s_3, p) = d(s_4, p) \wedge d(s_1, p) \leq d(s', p)\ \forall s' \in S \setminus \{s_1, s_2, s_3, s_4\}\}$.

The GVD is then given as $GVD(S) := \mathbb{R}^3 \setminus (VR(S) \setminus MA(S))$.

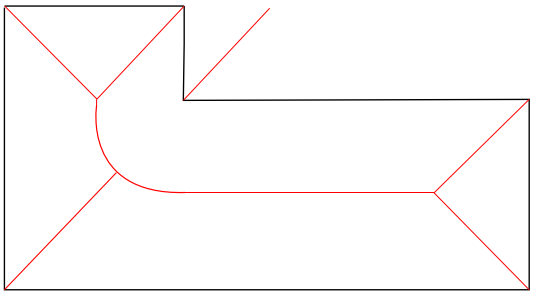

Fig. 11: The black lines represent an input site, the red lines are the corresponding medial axis MA.

Definition 5 (*Degree δ*)

We define the degree δ of a point $p \in VD$ which indicates the number of adjacent Voronoi cells. Thus:

$$\delta(p) \begin{cases} = 2, & \text{if } p \in VE \\ \geq 3, & \text{if } p \in VN. \end{cases}$$

And correspondingly for $p \in GVD$

$$\delta(p) \begin{cases} = 1, & \text{if } p \in MA \\ = 2, & \text{if } p \in VF \\ = 3, & \text{if } p \in VE \\ \geq 4, & \text{if } p \in VN. \end{cases}$$

The GVD is also the union over all medial axes, Voronoi faces, Voronoi edges and Voronoi nodes. The additional criterion $d(p_s, q_s) \geq \delta_{MA}$ results in a 'coarser' MA surface. Due to an 'uneven' surface - i.e. we mean with an uneven surface that the surface has a small and fine contour - too small δ_{MA} values result in an overflooded MA. Overflooded means that the GVD contains too many superfluous MA points, which makes it less meaningful. See Section 3.1.5 for a more detailed explanation and a formal discussion.

The ordinary VD can be computed efficiently, for example with a sweep line algorithm (Fortune, 1987). This generally does not hold for the GVD (Boissonnat and Teillaud, 2006). Since we are focusing on industrial data with millions of triangles, we consider its approximation. The main criteria with regard to practicability are calculation time and memory usage.

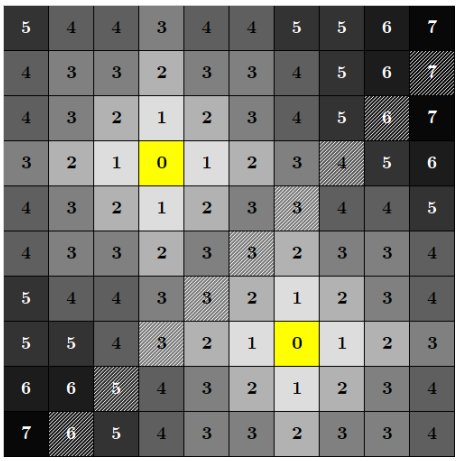

Fig. 12: The two-dimensional distance field shows for each point the minimal distance to the yellow-highlighted input data. The white hatched voxels represent the approximated VD.

The state-of-the-art algorithms for approximating the GVD can be divided into two main classes. The algorithms in the first class work with the exact input data but approximate the GVD. We can subclassify this in **render**- and **octree**-based algorithms. The second class approximates the input data with voxels and calculates the **distance field** (DF) that implicitly produces the GVD. Therefore, we briefly introduce distance fields.

3.1.1 Distance Field (DF)

For each point in a gird, a discrete distance field (DF) stores the nearest distance to a given input data (see Figure 12 for a two-dimensional example). The common underlying data structure for the DF is a two-dimensional array. A common strategy for calculating the distance field is the *wavefront propagation* (Ragnemalm, 1992). The algorithm starts with the input data and distance 0. The algorithm then recursively propagates the new distance to its neighbors. The VD can be obtained when different wavefronts meet each other. Approximating the VD with a DF approach is fast, but a DF requires cubic memory usage in the three-dimensional case.

3.1.2 Rendering and Octrees

Octree

The main idea starts with one octree, which includes the whole scene (Lavender *et al.*, 1992). The octree recursively subdivides itself into eight descendant octrees if it includes more than one site or a neighboring octree has a different nearest site. The reason is the following: if a cuboid c of the octree includes only one site and all neighbor cuboids have the same nearest site as c a finer representation of c with more octrees will not result in a finer GVD. Therefore, the octree subdivides only if it results in an improved resolution. This proceeding terminates if the desired resolution is achieved. It results in a finer resolution at the Voronoi boundaries and a coarser one inside the Voronoi cells. The octree represents the GVD with all leafs which have more than one nearest site. Whenever an octree is subdivided into eight smaller ones, however, the nearest site must be calculated for each descendant. This results in numerous distance queries, which makes this approach too slow for large input data.

Rendering

Render-based approximation algorithms divide the scene into equidistant 2D slices, calculate the GVD for each slice and stitch them together to create the three-dimensional GVD (Hoff *et al.*, 1999). The GVD for one slice is calculated as follows: For each triangle of each input mesh, render the uniquely colored graph of a specific distance function that is a hyperbolic function for a point, a cone for a line and a plane for the inner triangle (see Figure 13 for a graphical representation). The appearance of the functions depends on the distance between the site (point, line or inner triangle) and slice. So if a point lies on the current slice, the distance is 0, which means that the hyperbolic distance function is a cone. As the distance between the point and slice increases, the hyperbolic function becomes wider. Now render all the distance functions (see Figure 14 for a graphical representation). The GVD for the slice is extracted as follows: The distance functions of the same mesh have the same coloring and distance functions from different meshes have a different color. Therefore, the Voronoi regions of each mesh can be seen in the rendered image - as shown in Figure 14. However, rendering the graph of one of the distance functions requires a facetted version of this graph. This makes the approach too time-consuming for input containing millions of triangles.

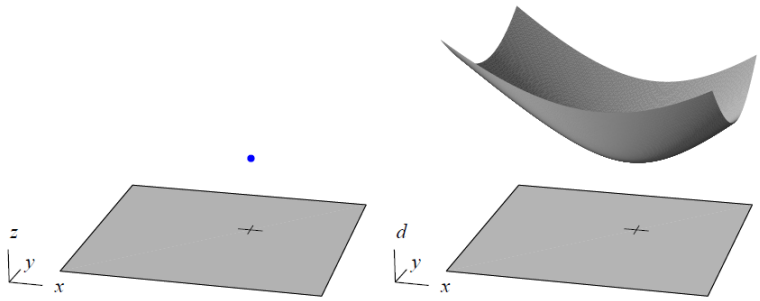

Fig. 13: A point (left) and the corresponding hyperbolic distance function (right) (Hoff *et al.*, 1999). The current slice is visualized via the plane.

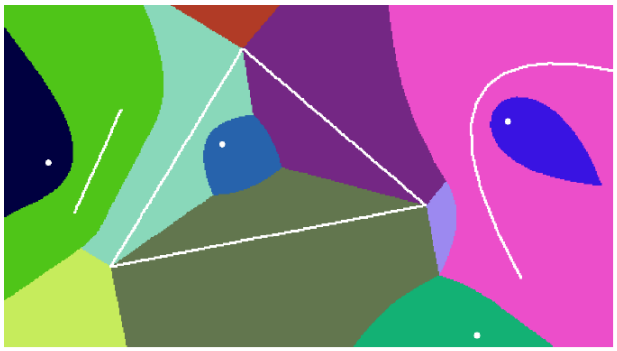

Fig. 14: The rendered distance functions of four points, a line, a triangle and one freehand line. (Hoff *et al.*, 1999).

3.1.3 Related Work

Over the past few decades, many approaches for approximating the GVD have been presented. We would now like to provide a comprehensive analysis of the related works covering the aforementioned render-, octree- and distance field-based approaches.

The **render-based** approach from Hoff *et al.* (1999) is used for many motion planning algorithms (Foskey *et al.*, 2001), (Garber and Lin, 2002), (Geraerts, 2010). Based on this work, the authors from Strzodka and Telea (2004) present an approximation for the distance field using arbitrary metrics for 2D data. Hsieh and Tai (2005) offers a number of optimizations for the GVD visualization. However, these works cannot solve the problem of the massive increase in runtime for complex input data.

An extension of the basic work from Lavender *et al.* (1992) on **octrees** is presented in Baston and Celes (2008). With the calculated octree, they create a polyhedral representation of the approximated GVD. In each propagation step, the algorithm

links neighboring octrees that are part of the GVD. A polyhedral representation of the GVD can therefore easily be obtained via the linked leaves of the octree. However, it can still take a long time to calculate the octree because the nearest neighbor query at each subdividing step is time-consuming. The focus in Edwards *et al.* (2015) is on computing the GVD for closely spaced objects. The octree data structure is optimized for this situation and works on arbitrary 3D meshes. This approach cannot overcome the problem of excessive runtime associated with the nearest neighbor query at each subdividing step.

For the related work on **distance fields** up to 2006, we refer to the extensive survey conducted by Jones *et al.* (2006). The main techniques use distance templates, space scanning (Danielsson, 1980) and different voxel propagation methods (Ragnemalm, 1992).

Another propagation algorithm for calculating *generalized* distance fields is the *fast marching* algorithm (Tsitsiklis, 1995). The fast marching algorithm is a numerical method which was designed to approximate the boundary value problem of the *Eikonal equation* $|\nabla u(x)| = 1/f(x)$ with $u(x) = 0$ for $x \in \partial\Omega$. This can be interpreted as time that different spreading sources (wich are on the boundary $\partial\Omega$) need in order to reach certain points, e.g. light and sound. We indicate this time for a voxel v as t_v and the source point that reaches v after time t_v as the starting voxel from v. The spreading speed of the sources can be described with different arbitrary functions f, i.e. each source has its own spreading function. The fast marching algorithm acts as described in the following: The starting voxels at time 0 propagate to all the neighbors, which are stored in a priority queue Q. Q orders all neighbors by their time t. If a voxel v propagates to a neighbor n, the followings steps are performed. Using the spreading function f, calculate the time that the starting voxel from v needs to propagate to n. If n was already visited, update the minimal time t_n of n. If the starting voxel from v delivers a smaller t_n than the current one, therefore update t_n and inherit the starting voxel from v to n. In addition, update the position from n in the priority queue. If n was not already visited, set the time needed t_n and the starting voxel and insert n in the priority queue q according to time t_n. If v has propagated to all neighbors, remove v from Q. Continue with the neighbor in Q that has the smallest t. The algorithm terminates if Q is empty. The fast marching algorithm calculates a common distance field by setting the spreading speed to all sources equally and constantly, i.e. all distance functions are the Euclidean distance function. However, inserting or updating a neighbor in the priority queue always needs logarithmic time. The fast marching

algorithm therefore has $\mathcal{O}(N \log N)$ calculation time, with N indicating the number of voxels in the grid. In summary, this method is not useful for our case because we do not need to consider arbitrary spreading functions f and the time is with $\mathcal{O}(N \log N)$ slower than our approach which has $\mathcal{O}(N)$.

Following the aforementioned survey, a number of GPU-based propagation methods were introduced (Rong and Tan, 2006), (Guo *et al.*, 2011), (Schneider *et al.*, 2009), allowing propagation be performed in parallel, which significantly increases speed. The authors from Cao *et al.* (2010) present a divide-and-conquer algorithm on the GPU. A CPU-based propagation algorithm also exists that focuses on fewer distance calculations (Velic *et al.*, 2009). All of the presented algorithms, however, still store the entire distance field. Yuan *et al.* (2011) present a memory-saving version of the *Jump Flooding Algorithm* (Rong and Tan, 2006). In the distance field, only the nearest site is stored as a short integer instead of a pointer to the nearest site. This yields a memory-saving factor of 6. However, this depends on the programming language and still cannot eliminate the cubic memory usage of distance fields.

3.1.4 Calculating the GVD

Main idea

The main idea behind our solution is to use the speed advantage of a propagation method without storing the complete distance field. The basic task during the propagation process is to find and check neighbors. In a complete distance field with a 3D array representation, this can be performed with a simple index shift. To compensate for the lack of a complete 3D array, we introduce our new data structure Voronoi voxel history (VVH). The VVH stores the voxels from the last few propagation steps in multiple hash tables. The access time of a 3D array is constant, which generally also holds for the VVH.

Preliminaries

The center of a voxel v is positioned at $v.pos$ in a $\mathbb{Z}^3$ grid. We use the Euclidean metric and define the neighborhood $N(v)$ of a voxel v as the union of the 26-neighborhood and the voxel itself. The discrete distance $d : \mathbb{R}^3 \times \mathbb{R}^3 \to \mathbb{N}_0$ between two vectors $a, b \in \mathbb{R}^3$ is defined as $d(a, b) := \lceil \|a - b\|_2 \rceil$ and the discrete distance of two voxels v_1, v_2 as $d(v_1, v_2) := d(v_1.pos, v_2.pos)$.

The algorithm

Our algorithm works in two steps. In the first step we use the technique from Fang and Chen (2000) to voxelize the scene with a voxel resolution of $\lambda > 0$. We render the scene in 2D slices, each of width λ, and interpret each colored pixel as a voxel that belongs to a site. Different colors represent the different input sites. We call the colored voxels *startingVoxels*. These are stored in a first hash table.

The second step is the propagation algorithm. We start with the startingVoxels and radius 0 and propagate the voxels in discrete, radius-increasing steps to their neighbors (for a graphical representation see Figure 15 and on page 34 Figure 20). This means that the radius r is both distance (measured in voxel size λ) and time step. When a neighbor is visited for the first time the propagating voxel forwards the information about its starting voxel (*voxel.start*), the distance (*voxel.dist*) to its starting voxel and the nearest site to which it belongs (*voxel.site*) to the neighbor. If the neighbor was already visited it is a candidate for the GVD.

We will now provide a detailed description of the procedure. The pseudocode also needs to be considered.

Class 1 VoronoiVoxelHistory

```
1: Queue<HashTable<Voxel>> data
2: function void updateVVH
3:     data.dequeue()                                   ▷ delete oldest voxels
4:     data.enqueue()                                   ▷ add empty neighborVoxels
5: function Voxel findVoxel(x, y, z)
6:     for 1 ≤ i ≤ 5 do
7:         if voxel(x, y, z) ∈ data[i][hashkey(x, y, z)] then   ▷ search in hash table
8:             return voxel
9:     return NULL                                      ▷ voxel is not stored
```

Voronoi voxel history (VVH): Class 1 is our voxel-managing data structure. The variable *data* is a queue of 5 hash tables (see Theorem 1). The hash table at position $1 \leq i \leq 5$ stores for a given radius $r \in \mathbb{N}_0$ all voxels with distance $r + i - 4$. We call the hash table at index $i = 4$ *currentVoxels* and the hash table at index $i = 5$ *neighborVoxels* (for a graphical representation see Figure 15 a)). The hash tables use separate chaining for collision resolution. A hash key for a voxel at position $(x, y, z) \in \mathbb{Z}^3$ which, among others, worked well for our application is $(x^3 + y^2 + z + xyz)$ mod $hashTable.size()$.

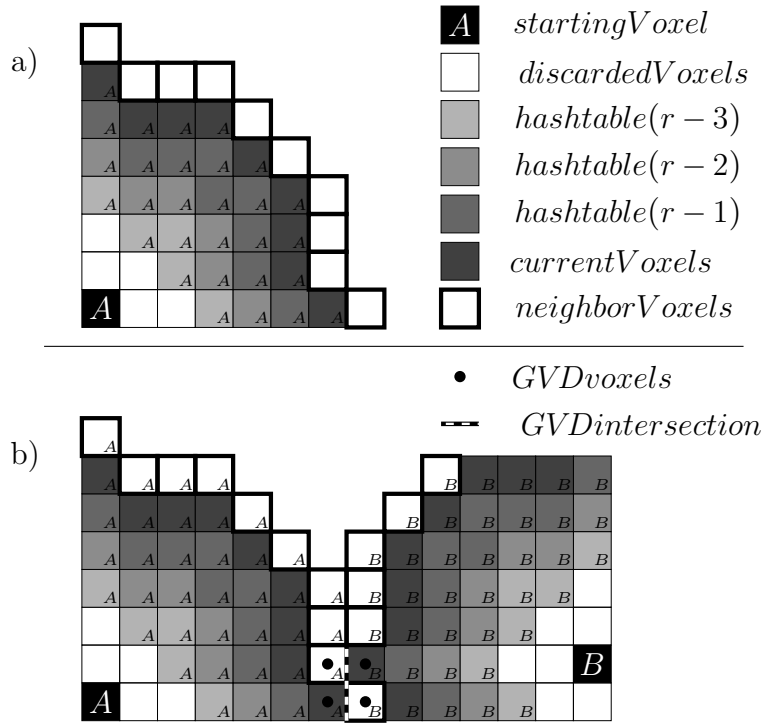

Fig. 15: a) An illustration of the data structure VVH. The A/B in the lower right corner of the voxels symbolizes the starting voxel ($voxel.start$). The $discardedVoxels$ are not stored anymore and the $neighborVoxels$ are stored after the propagation step. b) The VVH data structure after the propagation step from the $currentVoxels$. The $neighborVoxels$ are now stored in the VVH. The $GVDvoxels$ result from voxels with different starting voxels (A and B) meeting during the propagation. The dashed line $GVDintersection$ between the $GVDvoxels$ represents the non-volumetric boundary of the Voronoi cells.

At the end of one propagation step, the updateVVH function (lines 2 - 4) clears the latest voxels (line 3), i.e. the voxels with distance $r-3$, and inserts a new $neighborVoxels$ hash table (line 4). This represents an "aging" of 1 for all hash tables. The findVoxel(x, y, z) function searches for the voxel with the input coordinates (x, y, z) in the 5 hash tables of *data* and returns it. If the voxel is not stored yet it returns NULL.

calculateGVD: Algorithm 1 takes the $startingVoxels$ from the voxelization step as its input. We initialize the VVH and the GVD(line 1, 2), set the $currentVoxels$ to the $startingVoxels$ and the radius to $r = 0$ (lines 3, 4). The computation of the GVD is complete when there are no more $currentVoxels$ in the VVH. The size of the hash table $neighborVoxels$ is set to the number of voxels contained in $currentVoxels$ (line 6). Since a current voxel has at most 27 neighbors, this ensures a constant average query time for the new hash table. The following loop (lines 7, 8) propagates every voxel in $currentVoxels$, filling the $neighborVoxels$ hash table. When the propagation

stops, propagation step r is complete, and we update r and the VVH for the next step (lines 9, 10). Finally, we return the GVD in line 11.

Algorithm 1 GVD calculateGVD($startingVoxels$)

```
1:  GVD ← new GVD
2:  vvh ← new VoronoiVoxelHistory
3:  vvh.currentVoxels ← startingVoxels
4:  r ← 0                                          ▷ Initialize the radius
5:  while vvh.currentVoxels.size() > 0 do
6:      vvh.setNeighborVoxelsSize()
7:      for all voxel ← vvh.currentVoxels do
8:          propagateVoxel(GVD, vvh, voxel, r)
9:      r ← r + 1
10:     vvh.updateVVH()
11: return GVD
```

Algorithm 2 void propagateVoxel($GVD, vvh, voxel, r$)

```
1:  if d(voxel, voxel.start) > r then   ▷ Check if the voxel would propagate too early
2:      vvh.addNeighbor(voxel)                      ▷ Add the voxel to neighbors
3:      return
4:  for all (Δx, Δy, Δz) ∈ {−1, 0, 1}^3 do
5:      nPos ← voxel.pos + (Δx, Δy, Δz)
6:      if d(nPos, voxel.start.pos) < r + 1 then   ▷ Check if nPos was visited before
7:          continue
8:      n ← vvh.findVoxel(nPos)
9:      handleNeighbor(GVD, vvh, voxel, n, nPos)
```

propagateVoxel: Algorithm 2 propagates a voxel to its neighbors. In lines 1 - 3, we fix the *overpropagation*. To ensure that all nearest sites are set correctly, it can may be the case that a neighbor is inserted into $neighborVoxels$ too early. See the next paragraph for a detailed explanation of the *overpropagation*. If the propagating voxel has a greater distance to its starting voxel (it was *overpropagated*) than the current propagation step r (line 1), we add the voxel to the neighbors (line 2), so that the voxel can propagate in the next step, and return (line 3). Otherwise, the propagation loop in line 4 goes ahead. The position $nPos$ of the neighbor is set in line 5. If the distance

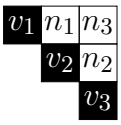

Fig. 16: A two-dimensional example showing the need for the overpropagation. The black voxels v_1, v_2, v_3 are starting voxels. Voxels n_1, n_2, n_3 are voxels that have not yet been visited.

d from the neighbor position $nPos$ to its starting voxel ($voxel.start$) is smaller than $r+1$, this means that we already visited this neighbor. We thus discard this neighbor (lines 6, 7). Otherwise we search for the neighbor n at position $nPos$ in $data$ and call handleNeighbor (lines 8, 9).

overpropagation: In each propagation step all, we add new neighbors to the $neighborVoxels$. This ensures, that each voxel has the nearest assigned starting voxel. It is possible, however, that a neighbor n has a distance to its starting voxel that is greater than $r+1$. Consequently, in the next propagation step $(r+1)$, n is part of the $currentVoxels$ but its distance does not match the time step $r+1$. We explain this case using the two-dimensional example from Figure 16. The nearest voxel of n_3 is v_2 with the distance $d(n_3, v_2) = \sqrt{2}$. If v_2 propagates in the first step $(r = 0)$ to n_3 the correct nearest voxel is set. However, the integer distance is $d(n_3, v_2) = \lceil\sqrt{2}\rceil = 2 = r+2$. Therefore, the voxel n_3 should not propagate in the next step $r+1 = 1$. After explaining why a neighbor can be inserted in $neighborVoxels$ too early we explain why it is necessary to perform this procedure for a correct propagation process. Let's assume that we ignore the voxel n_3 in the first propagation step. Depending on the voxel propagation order it is possible that we have $n_1.start = v_1$ and $n_2.start = v_3$. This means that in the next propagation step, we obtain $n_3.start = v_1$ or $n_3.start = v_3$, which are not the nearest voxels.

Algorithm 3 void handleNeighbor($GVD, vvh, voxel, n, nPos$)

1: **if** $n ==$ NULL **then** ▷ the neighbor was not already visited
2: $vvh.addNeighbor(voxel.start, nPos)$
3: **else if** $voxel.site \neq n.site \vee \delta_{MA} < d(voxel.start, n.start)$ **then**
4: $GVD \leftarrow GVD \cup \{voxel, n\}$
5: $n.start \leftarrow \underset{v \in \{voxel.start, n.start\}}{\arg\min} \|n.pos - v.pos\|_2$ ▷ update the nearest startingVoxel

handleNeighbor: We first check whether the neighbor was already visited (line 1). If it was, create it and insert it in the $neighborVoxels$ hash table of the VVH (line 2).

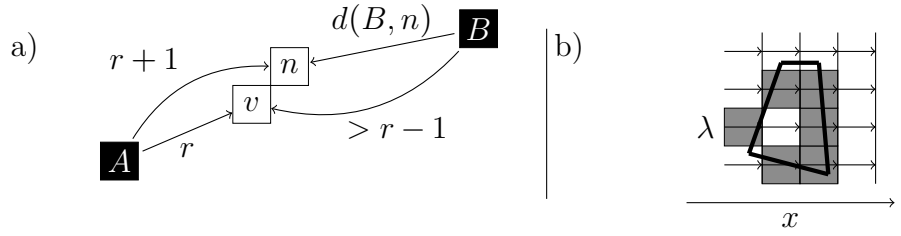

Fig. 17: a) Graphical representation of the voxel v which propagates to its neighbor n. b) The result of the render-based voxelization in $+x$ direction with voxel size λ. There are 4 slices with width λ and 4 pixels per slice, where each slice is visualized by a vertical line and the pixels by the arrows. If an arrow hits the mesh, the corresponding pixel is colored accordingly.

If it was not, two options are available. First, if the nearest sites of the voxel and its neighbor differ (first condition in line 3) the condition for a GVD voxel is satisfied and we add both voxels to the GVD (line 4). Second, if the nearest sites (*voxel.start.site*, *neighbor.start.site*) are equal and the distance of their starting voxels is greater than a given parameter (second condition in line 3), we broaden the GVD by also adding the two voxels lying on the medial axis of the (large) input site. Finally, we update the nearest starting voxel *n.start* of n (line 5).

From the approximate GVD obtained in this way we can at the end derive a more specific approximation of the GVD by considering the intersections (see the dotted line *GVDintersection* in Figure 15 b), which can be squares, lines or points in 3D) of all the added voxel pairs *voxel* and n.

Analysis

It is important to ensure that the propagation waves from different starting voxels (as shown in Figure 15 b) for the 2D case) do not propagate through each other. We show in Theorem 1 that the VVH stores all the voxels needed for this. A visual representation is shown in Figure 17 a).

Theorem 1

The VVH needs 5 *hash tables for a correct propagation process.*

Proof: Let $r \in \mathbb{N}_0$ be the current propagation step and v a current voxel with $v.start = A$ that propagates to its neighbor n. This means that $d(A, v) = r$ and $d(A, n) = r + 1$. Now assume that n was already visited with $n.start = B$.

We want to prove that the information that n was already visited is still stored in the VVH.

Since v has A as its nearest starting voxel it follows that $d(B,v) > r-1$. This leads to

$$r-1 < d(B,v) \leq d(B,n) + d(v,n) \leq d(B,n) + \left\lceil \sqrt{3} \right\rceil .$$

We derive $d(B,n) > r-3$.

Under the assumption that we could compute the numerically correct distance between voxels it would be sufficient to store the $neighborVoxels$ $(r+1)$, the $currentVoxels$ (r) and the two previous hash tables $r-1$ and $r-2$. To properly handle numerical inaccuracies, the hash table $r-3$ also needs to be stored. ■

Next, we prove the error-bound for the approximation of our GVD. We assume that the voxel size λ is sufficiently small so that every site and substantial subgeometry is intersected by the rasterization.

Theorem 2

Let $\lambda > 0$ be the voxel size. Then the approximation of the GVD has an error of at most 2.232λ.

Proof: With a voxel size of λ and the render process in different directions (as described in Fang and Chen (2000)) we can guarantee that the voxelization produces an error of at most $\lambda/2$ (for a graphical representation see Figure 17 b)).

Let $a, b \in$ GVD be two neighbored voxels with different starting voxels. We guarantee in Algorithm 2 (lines 1 - 3) that no voxel propagates too early. In addition, every voxel propagates to all its neighbors (Algorithm 2 line 6) its nearest starting voxel (Algorithm 3 line 5). Therefore, the discrete steps respect the Euclidean metric. So the exact GVD of the starting voxels *a.start* and *b.start* passes somewhere through the union of the voxels a and b. By setting the approximate GVD to the intersection of a and b (see the dotted line *GVDintersection* in Figure 15 b)) and with a space diagonal of length $\sqrt{3}\lambda$ for a voxel we have an error of at most $\sqrt{3}\lambda$.

This yields an overall error of at most $(1/2 + \sqrt{3})\lambda \approx 2.232\lambda$. ■

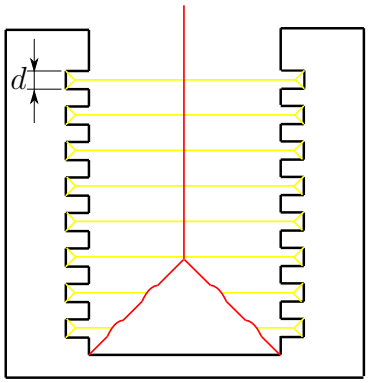

Fig. 18: The black lines represent an input site. The yellow lines represent the flooded and the red lines the 'important' MA outside of the input site. For simplicity we did not draw the MA inside of the site.

3.1.5 Experiments

In this section, we evaluate the medial axis for different δ_{MA} values looks like. In addition, we analyze the GVD for different voxel sizes λ. The focus for the GVD is on runtime and memory consumption. We compare the number of voxels used in our approach with the ones used by a complete distance field.

Medial axis

We discuss our modification $d(p_s, q_s) \geq \delta_{MA}$ of the MA in Definition 4. The modification requires for a MA point that the distance between the two corresponding points p_s, q_s is greater or equal as δ_{MA}. The original definition from Blum (1967) has no distance constraint d for the points p_s and q_s, i.e. our modification becomes the original definition by setting $\delta_{MA} = 0$. This works for simple shapes like those shown in the example in Figure 11. However, 'uneven' surfaces flood the MA as shown by way of example in Figure 18. Considering the background, that later parts should be disassembled along the MA, we call the yellow-highlighted flooded points *unnecessary* because these points cannot be used for successful motion planning. Furthermore, a worker who assembles the part also needs a minimum clearance during the assembly process because their hands must also pass the workspace along the disassembly path. Nevertheless, each value δ_{MA} that is greater than 0 can prevent useful parts of the MA. One example is the following (see Figure 30 on page 48 for a graphical representation): A screw can be inserted a few millimeters into a hole. A worker can easily assemble this screw, although the clearance is only a few millimeters. The unnecessary MA points disappear in the example shown when $\delta_{MA} = d + \epsilon$ and only the red lines remain.

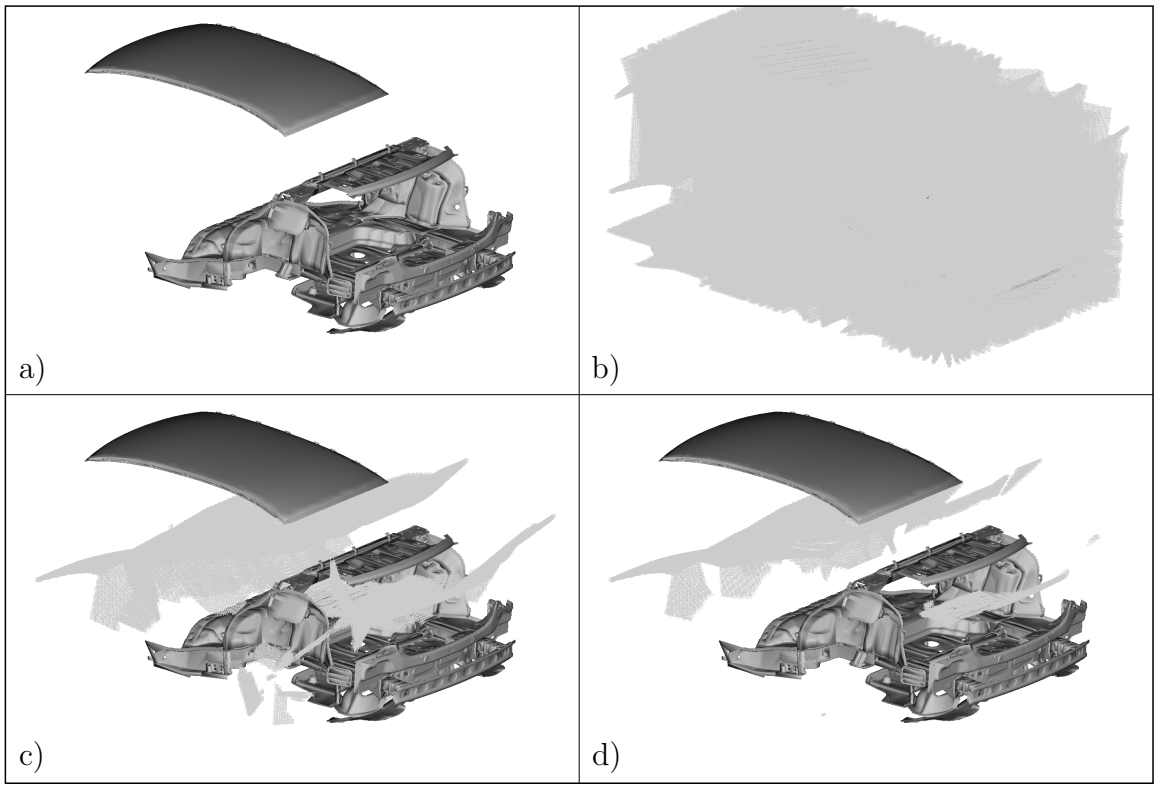

Fig. 19: The MA of the input site, seen in a), for different values δ_{MA}. b) $\delta_{MA} = 0$, c) $\delta_{MA} = 30$, d) $\delta_{MA} = 50$.

However, setting δ_{MA} too high will result in an empty MA. Depending on the data set, therefore, it is important to set a suitable value for δ_{MA}.

Our experiments with different δ_{MA} values for the given test data (see Figure 19 a)) revealed that 30 millimeters (c)) delivers the desired result - i.e. an MA that is slim and does not miss important surfaces for the subsequent motion planning phase. The MA for $\delta_{MA} = 0$, shown in b), is extremely flooded. The MA for $\delta_{MA} = 50$, shown in d), is too sparse. However, the right value for δ_{MA} and the optimal MA cannot be determined mathematically. We can therefore only choose a 'good' value for δ_{MA} in order to obtain a 'good' MA.

General Voronoi diagram

The results of the tests for the GVD calculation are for both data sets and with the different voxel sizes presented in Table 4. The needed voxels of our VVH grows generally more than just quadratically. This is because we need to store the voxels from the voxelized parts. The number of voxels for thin parts can increase cubically as shown in example Figure 21. Mid: The part is thinner than the resolution and therefore

Table 4: The evaluation of the GVD calculation.

data set	λ [mm]	GVD		DF		ratio
		t [s]	*#voxels*	t [s]	*#voxels*	#voxels
a)	20	8	490,000	1.5	750,000	1.54
	10	72	2,280,000	9	6,050,000	2.65
	5	525	13,580,000	67	48,420,000	3.56
	2	11,687	68,478,000	841	756,575,000	11.04
b)	20	25	1,080,000	6	1,510,000	1.39
	10	160	7,050,000	28	12,100,000	1.71
	5	1245	40,210,000	141	96,840,000	2.40

both wands are represented with the same voxel. Right: Each wand is represented by seperate voxels. Therefore, the number of voxels increase unproportional. This circumstance increases the number of needed voxels, i.e. in the three dimensional case one voxel gets subdivided into eight voxels which is a cubic increase of the number of voxels.

Figure 20 shows the propagation steps for data set a). We set the parameter δ_{MA} for the medial axis condition in line 3 of Algorithm 3 to 30 mm. Column "t" contains the overall running time of our implementation for voxelizing the scene and calculating the GVD. The voxelization step needs only a small percentage (<5%) of the overall calculation time. We can compute the GVD for the almost complete car in dataset b) with a high resolution of $\lambda = 5$ mm in less than 25 minutes. Furthermore, the test on data set a) with $\lambda = 20$ mm and t = 8 seconds shows that our approach is downscalable. Our algorithm focuses on very large and complex data sets but it can also be used for smaller data sets or coarser grids where the focus is on a fast calculation time.

The following two columns show the number of voxels needed by our GVD and the complete distance field (DF). Regarding the space consumption of a single voxel, a voxel in the VVH needs to store its position (*voxel.pos*). This is due to the unstructured ordering in the hash table and is not necessary for a distance field. But since in our application the GVD or DF computation is only a preprocessing step for the subsequent GVDG calculation, it is recommended to store for each voxel its starting voxel (*voxel.start*), position (*voxel.pos*), clearance (*voxel.dist*) and a list of neighboring voxels with different nearest sites. This means that the space consumption of a single voxel is almost identical for both data structures.

Therefore, the presented number of voxels for different resolutions in Table 4 shows the memory growth of our approach compared with the cubic growth of a complete

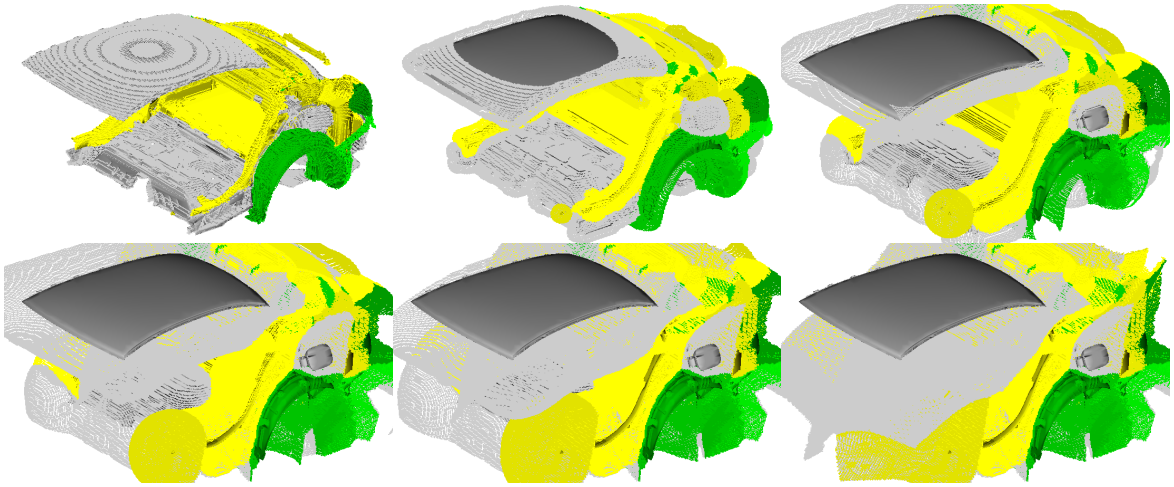

Fig. 20: The voxel propagation process with a voxel size of $\lambda = 10$ mm for data set a) from Figure 5. The chronological sequence is shown from the top left to bottom right.

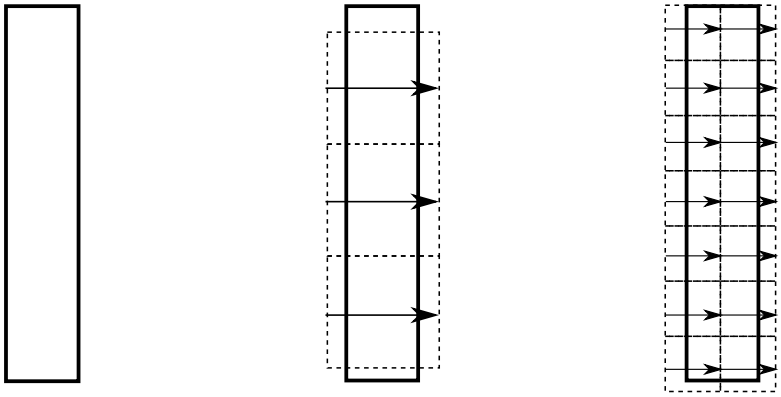

Fig. 21: A two-dimensional example of a voxelized part with two different resolutions. The arrows represent the render rays and the dashed boxes the voxels. Left: The part; Mid: The voxelized part; Right: The voxelized part with a resolution which is double that fine as from 'Mid'.

distance field. As a result, the "ratio" between the number of voxels for a complete distance field and for our approach increases as the voxels become finer. The need for a voxel-saving approach becomes apparent for data set a) with a voxel size of $\lambda = 2$ mm. Here the voxel ratio is already 11.04. Comparing the time required for both underlying data structures it can be seen, that the ratio between the VVH and DF increases as λ becomes finer. Using the VVH for $\lambda = 10$ mm takes 8 times longer than a calculation with a DF. In summary, the RAM-saving property of the VVH comes at the cost of a longer calculation time. If the required memory usage exceeds the given memory usage, however, the less memory-usage intensive variant should be used.

3.2 General Voronoi Diagram Graph (GVDG)

As shown in Geraerts (2010), finding a disassembly path on a GVD-based roadmap, called *general Voronoi diagram graph* (GVDG), is runtime-effective for finding short paths with sufficient clearance. We define the GVDG as follows.

Definition 6 (*General Voronoi diagram graph*)

Given $S := \{s_1, ..., s_n\}$ the input data which consists of n three-dimensional meshes and the corresponding GVD. We define the general Voronoi diagram graph $GVDG := (V, E)$ as an undirected graph where the set of nodes V as well as the edges E are part of the GVD.
An edge $e \in E$ is defined as a set of ordered voxels $e := \{v_1, ..., v_n\}$. We define the clearance c of an edge e as $c(e) := \min_{v \in e}\{d(v, v.start)\}$.

In the two-dimensional ordinary case (Definition 3) the Voronoi diagram graph (VDG) is the VD itself. The nodes V are the set of Voronoi nodes VN and the edges E are the Voronoi edges VE (see Figure 10). A point robot has no dilation and so moving a point robot along the VDG ensures a collision-free navigation. Based on this VDG, an optimal disassembly path for a point robot - e.g. the shortest disassembly path along the maximal clearance - can now easily be found with a Dijkstra query.

In our scenario, however, the GVDG does not generally deliver feasible disassembly paths (see Chapter 4 for a formal definition) for three-dimensional meshes but it provides estimated translational disassembly paths, so called *Voronoi paths (VPs)*, which can be sued to find feasible disassembly paths. Later (see Section 4.5), we use the VP as a translational basis for our 'Expansive Voronoi Tree' which samples along the Voronoi path and finds fast feasible disassembly paths.

Extracting a GVDG from a GVD is a complex task. The *straightforward approach* from Foskey *et al.* (2001) defines the GVDG as follows: Set the GVDG equal to the Voronoi nodes VN and Voronoi edges VE, i.e. $GVDG := (VN, VE)$. However, this approach leads to *isolated sites* which are defined as follows.

Definition 7 (*Isolated site*)

Given $GVDG := (V, E)$ a general Voronoi diagram graph and a set of input sites S. Then a site $s \in S$ is isolated *if the GVDG does not provide a path from the Voronoi cell $VC(s)$ to a body-in-white point. We say that a site s is represented in the GVDG if a voxel from the Voronoi cell $VC(s)$ is part of the nodes V.*

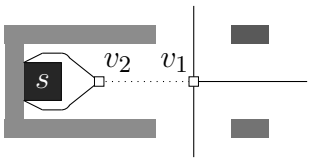

Fig. 22: A two-dimensional assembly that shows four sites and the corresponding GVD. The solid line represents the boundary of the Voronoi cells and the dotted line the medial axis. The voxels v_1 and v_2 have after Definition 5 degree $\delta(v_2) = 2$ and $\delta(v_1) = 3$.

A true isolated site *is isolated and there exists no feasible disassembly path for s to a body-in-white point. A* false isolated site *is isolated but there exists a feasible disassembly path to a body-in-white point.*

Using the GVDG as a basis for motion planning in the context of assembly sequence planning is better than improving only the motion planning process itself. If the GVDG can guarantee that there are no false isolated sites, all isolated sites are truly isolated. Therefore, no disassembly path exists for any of the isolated sites. For all these isolated sites, therefore, motion planning queries that will be inevitably unsuccessful can be saved, which greatly improves the runtime (see also Section 5.4 on page 89). This is why we broaden the requirement regarding a GVDG such that it should not contain false isolated sites.

To eliminate false isolated sites we analyze in which cases true and false isolated sites occur. A typical true isolated site is geometrically enclosed, e.g. if all doors and coverings in a car are closed, no inner part can be disassembled. The reasons for false isolated sites depend on how the GVDG is calculated. First, we analyze false isolated sites if the $GVDG := (VN, VE)$ is calculated with the aforementioned straightforward approach. For a graphical representation of an isolated site in the two-dimensional case with arbitrary input sites, see Figure 22. In the example shown, the Voronoi cell of s is isolated for two reasons: First, s has only one neighbor that borders on its Voronoi cell: second, there is no Voronoi node n on the Voronoi cell $VN(s)$ that could represent s in the GVDG, i.e. there is no VN on the boundary of the Voronoi cell $VC(s)$. Even assuming that a Voronoi node (e.g. v_2) exists there is only an MA edge but no Voronoi edge that connects the Voronoi cell with v_1. Therefore, two cases can be derived: (i) Not every Voronoi cell has a Voronoi node; (ii) considering only the Voronoi edges VE for connecting nodes, one misses important edges which are part of the Voronoi faces and medial axes.

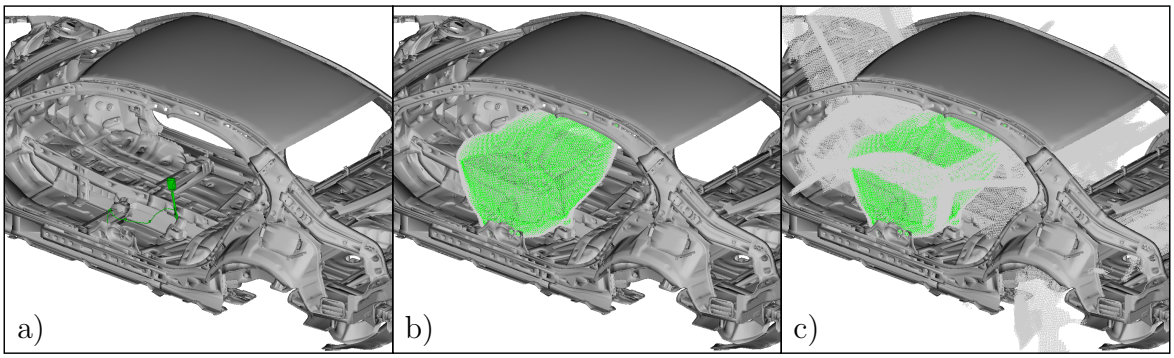

Fig. 23: The test data with the body in white (grey) and the assembled seat belt clip (green) is shown in a). The VC(seat belt clip) (green) is shown in b) and in addition the MA(body in white) in c).

A three-dimensional isolated site with the same issues as the two-dimensional example is shown in Figure 23. The Voronoi cell for the seat belt clips (see a)) has no Voronoi node, nor is there a Voronoi edge connecting the Voronoi cell to the outside (see b)). Thus, the straightforward approach could not deliver a Voronoi path for the seat belt clip. The medial axis of the body-in-white connects the Voronoi cell to all directions of the space (see c)). In this case, the medial axis of the body-in-white could provide an edge that connects the Voronoi cell and a body-in-white point.

As shown in Section 3.1.5, however, the medial axes need to be coarsened by setting the value $\delta_{MA} > 0$, which means that some parts of the MA are prevented. These parts of the MA that have been prevented may contain possible edges, so a site can be isolated even if there is a node and the MA is considered. This happens in real-world scenarios, e.g. for inserted screws and nuts. Summarized, taking into account the medial axes and Voronoi faces for the edges E only partially solves the issue of false isolated sites, because there are still cases where these approaches fail.

In the following, we present two novel approaches for calculating the GVDG and compare them with each other. Both approaches are propagation algorithms that start with a defined set of nodes V and propagate along the GVD to create edges. The two approaches differ primarily in terms of the choice of the nodes V. One approach starts with voxels on the Voronoi cells and propagates to the body-in-white points while the other approach starts with the body in white points which propagate into the inside of the GVD. Roughly said, one approach propagates from the inside to the outside and the other one vice versa. We will therefore first describe the general propagation algorithm, which is technically the same for both algorithms. As described, dealing with the problem of missing nodes (i) and taking into account Voronoi faces and medial

Table 5: The three different approaches for calculating the $GVDG := (V, E)$: Straight forward approach (SF), inside to outsight (ITO) and outside to inside (OTI). The underlying $GVD := (VN, VE)$ and the body-in-white points (BIWP) are the basis for the different approaches. The sets R_1 and R_2 each represent a set of nodes extracted from the respective approach.

criteria	SF	ITO	OTI
use prop algorithm	x	✓	✓
$startingVoxels$	x	$VN \cup R_1$	BIWP
V	$VN\cup$ BIWP	$VN\cup$ BIWP$\cup R_1$	BIWP$\cup R_2$
E	VE	propagation	propagation

axes (ii) can still result in false isolated sites. After describing the two propagation methods, therefore, we will then look at a $handleFalseIsolatedSites$ algorithm that 'connects' the remaining false isolated sites.

Table 5 provides an overview of the straightforward approach and the two approaches that are presented in the following subsections. The ITO and OTI approaches use the same propagation algorithm and differ only in how the $startingVoxels$ for the underlying propagation algorithm and the nodes V are chosen. The edges result from the propagation algorithm. The SF approach has no further calculation: it uses only the Voronoi nodes VN and the Voronoi edges VE for its GVDG. This table is designed to provide only a brief overview. How the specific variables are set and how the propagation algorithm works are described in the following.

3.2.1 GVDG Propagation Algorithm

In this subsection, we describe our GVDG voxel propagation algorithm. The algorithm propagates along a given GVD to create nodes and edges of the GVDG. However, the algorithm described is not yet complete, in the sense that it directly computes a GVDG. It is not determined how the $startingVoxels$ are set and under which circumstances an edge is created. In Section 3.2.2 and Section 3.2.3, we present two different approaches; the *inner to outer (ITO)*- and the *outer to inner (OTI)* propagation algorithm. They use the propagation algorithm from this subsection but the $startingVoxels$ and the edges are different. This subsection is therefore the basis for the following two subsections.

The algorithm for the GVD calculation on page 27 can be roughly structured as follows: (i) Voxelize the input sites and use them as $startingVoxels$; (ii) use a metric-respecting

neighbor propagation in the three-dimensional grid; (*iii*) insert two voxels in the GVD if these voxels encounter one another.

The propagation algorithm for the GVDG works with the same basic steps, only with some minor changes: (*i*) The $startingVoxels$ are pre-defined nodes; (*ii*) change the feasible underlying propagation space from the complete three-dimensional grid to the GVD; (*iii*) create an edge via backpropagation between two nodes if the corresponding propagating voxels encounter one another.

In Section 3.2.2 and Section 3.2.3, we show two applications of the GVDG propagation algorithm. They mainly differ in the choice of the $startingVoxels$ and the $existEdge$ function. However, this results in significantly different GVDGs.

To realize the backpropagation for the GVDG calculation, each voxel $v \in$ GVD stores the voxel that propagated to it ($v.previous$) and its startingNode ($v.startNodeGVD$) from the GVDG propagation process. Both are NULL by default. For each starting voxel, we set $v.previous$ = NULL and $v.startNodeGVD = v$. The propagation algorithm for calculating the GVD needs just some minor changes which are described below. Therefore, the description is high-level and we point out only the changes. For a graphical representation of the propagation algorithm, see Figure 24. We marked the elements that have been changed or are new. The changes are discussed in the squared brackets and also shown as black text in the pseudocode, i.e. the unchanged parts are light gray.

Algorithm 4 **GVDG** calculateGVDG(V, GVD)

```
1:  GVDG ← new GVDG
2:  vvh ← new VoronoiVoxelHistory
3:  vvh.currentVoxels ← V
4:  r ← 0                                              ▷ Initialize the radius
5:  while vvh.currentVoxels.size() > 0 do
6:      vvh.setNeighborVoxelsSize()
7:      for all voxel ← vvh.currentVoxels do
8:          propagateVoxelAlongGVD(GVDG, GVD, vvh, voxel, r)
9:      r ← r + 1
10:     vvh.updateVVH()
11: return GVDG
```

calculateGVDG: Changed: [Algorithm 4 gets some nodes V as input instead of the $startingVoxels$ from the voxelization step]. It initializes the [GVDG] and the VVH in lines 1 - 3. The main loop in line 5 terminates when there are no more voxels that can

propagate to their neighbors. The size of the current neighbors' voxels hash table is set in line 6. Each voxel then propagates to its neighbors (lines 7, 8) [which are part of the GVD]. At the end of each iteration of the while loop, the radius increases and the hash table indices are shifted (lines 9, 10). [In the end, we return the calculated GVDG.]

Algorithm 5 **void** propagateVoxelAlongGVD($GVDG, GVD, vvh, voxel, r$)

```
1: if d(voxel, voxel.start) > r then ▷ Check if the voxel would propagate too early
2:     vvh.addNeighbor(voxel)                                  ▷ Add the voxel to neighbors
3:     return
4: for all (Δx, Δy, Δz) ∈ {−1, 0, 1}^3 do
5:     nPos ← voxel.pos + (Δx, Δy, Δz)
6:     if d(nPos, voxel.start.pos) < r + 1 then  ▷ Check if nPos was visited before
7:         continue
8:     n ← gvd.findVoxel(nPos)
9:     handleNeighborGVDG(GVDG, vvh, voxel, n, nPos)
```

propagateVoxelAlongGVD: Algorithm 5 propagates a voxel to its neighbors. If the to propagating voxel has a greater distance to its starting voxel than the current propagation step r (line 1) we add the voxel to the neighbors (line 2), so the voxel can propagate in the next step, and return (line 3). Otherwise, the propagation loop in line 4 goes ahead. The position $nPos$ of the neighbor is set in line 5. If the distance d from the neighbor position $nPos$ to its starting voxel ($voxel.start$) is smaller than $r + 1$, this means that we already visited this neighbor and so we discard this neighbor (lines 6, 7). Changed: [Line 8 changes from $n \leftarrow vvh.findVoxel(nPos)$ to $n \leftarrow gvd.findVoxel(nPos)$. With respect to the underlying data structure $gvd.findVoxel(nPos)$ searches in a three-dimensional distance field or in a hash table for the voxel at position $nPos$. This means, that n is NULL, if the GVD does not have any voxels stored at the respective position. Line 9 calls the new handleNeighborGVDG function which is described next.]

Algorithm 6 **void** handleNeighborGVDG($GVDG, vvh, voxel, n, nPos$)

```
1: if n == NULL then                                          ▷ Check if n ∈ GVD
2:     return
3: if n.startNodeGVD == NULL then                             ▷ Propagate to n
4:     n.previous = voxel
5:     n.startNodeGVD ← voxel.startNodeGVD
6: if !existEdge(voxel, n) then
7:     E ← E ∪ createEdge(voxel, n)
```

handleNeighborGVDG: The *handleNeighborGVD* function from Algorithm 3 is replaced by *handleNeighborGVDG* in Algorithm 6. [If neighbor n is not part of the

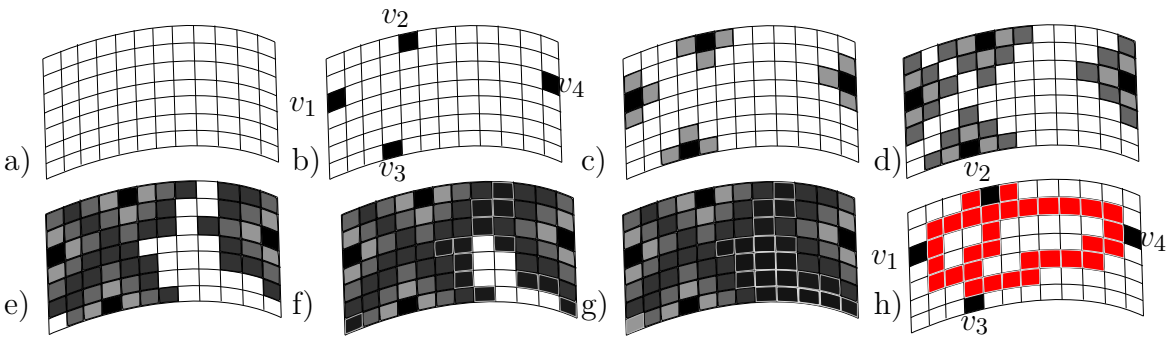

Fig. 24: A subset of a GVD is shown in a). The exemplary nodes v_1, v_2, v_3, v_4 in b) are black highlighted. The propagation process along the GVD is shown in c) - g). The resulting, red highlighted edges which connects the nodes are shown in h).

GVD (line 1) the function returns (line 2). If n was not already visited (line 3), *voxel* propagates and inherits the propagation information to n (line 4, 5). In line 6, check if an edge already exists. If not, the edge is created via backpropagation (line 7). Note: The *existEdge* function depends on the underlying propagation algorithm and so is described later in more detail.]

3.2.2 Inside-to-outside propagation (ITO)

We describe our intuitive extension of the described straightforward approach, the *inner-to-outer propagation (ITO)*, where we also consider Voronoi faces and medial axes as part of the edges. A graphical representation of the Voronoi nodes and the propagation process is shown in Figure 25. This approach is an application of the propagation algorithm in Section 3.2.1, i.e. we use the propagation algorithm and explain how it behaves if we set the nodes as follows. Set the nodes V, just like in the straightforward approach, so that they are equal to the Voronoi nodes from the GVD. For each site s that has no Voronoi node, insert a random voxel from its Voronoi cell $VC(s)$ to V. In this way, we ensure that each site has a representing voxel in the nodes V. With the nodes V, start the aforementioned, modified GVDG propagation from Algorithm 1. The *existEdge* function in Algorithm 7 excludes loops (lines 1, 2) and multiple edges between the same starting nodes (lines 3, 4).

The ITO approach is an improvement on the straightforward approach in two ways: First, it ensures that each site has a voxel from its Voronoi cell in the nodes V. Thus, each site is represented by the GVDG. The use of the aforementioned GVD-based

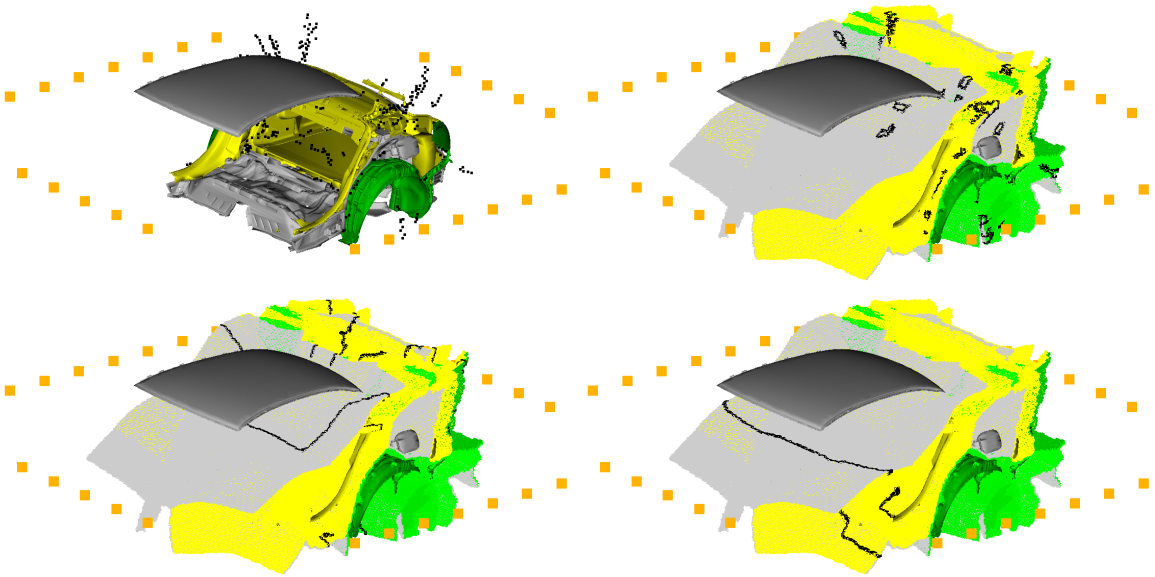

Fig. 25: The OTI propagation along the GVD. The black-highlighted voxels in the first image show the starting nodes. The black-highlighted voxels in the other images represent the current wavefront.

Algorithm 7 bool existEdge($voxel, neighbor$)

```
1: if voxel.startingNodeGVDG == neighbor.startingNodeGVDG then
2:     return true
3: if (voxel.startingNodeGVDG, neighbor.startingNodeGVDG) ∈ E then
4:     return true
5: return false
```

propagation algorithm means that Voronoi faces and medial axes are also considered for calculating the edges. Due to the propagation algorithm, which also considers Voronoi faces and medial axes for calculating the edges E the GVDG has significantly more edges than a GVDG created with the straightforward approach (see Figure 29). These edges also connect before unconnected body-in-white points which results in a significantly more varied and meaningful GVDG.

Although each component is represented by a node, it may be the case that this representation is not well suited and later results in suboptimal Voronoi paths. According to Definition 4, a Voronoi node arises if it has at least a degree δ of 4. Thus, an appropriate representation of nodes on the complete Voronoi cell is highly dependent on the number of sites in the neighborhood of a site. For example, a large Voronoi cell needs more than one Voronoi node for a good representation. The Voronoi nodes shown in the example in Figure 26 on the left do not properly represent the Voronoi cell. There are no nodes that are positioned in the direction of the rear end. A subsequent Dijkstra

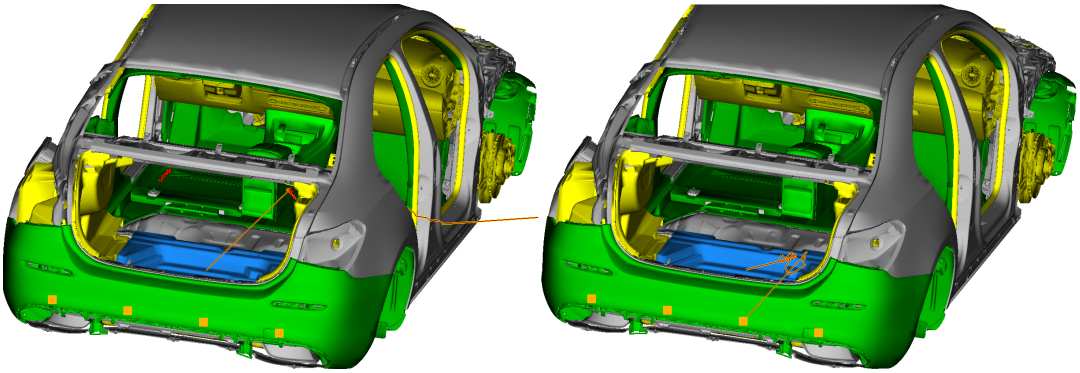

Fig. 26: The blue-highlighted part and its Voronoi nodes (red squares) on the left. Voronoi path lengths: left, 2000 mm; right, 1700 mm.

query, therefore, delivers the shown Voronoi path as the shortest path. However, the Voronoi path shown on the right, which is extracted with the *OTI* GVDG algorithm (presented in Section 3.2.3), is shorter.

Another minor disadvantage of the edges created is that a Voronoi path from a node to a body-in-white point has a zig-zag pattern, making the Voronoi path longer than it should be. This occurs because every node has only a straight connecting edge to the nodes that it reaches with the propagation algorithm, e.g. nodes v_1 and v_4 have no direct connection in the example in Figure 24, page 41. Thus, a connection between v_1 and v_4 requires a the detour via v_2 or v_3.

In summary, this approach is superior to the straightforward approach. It can still be further optimized in some ways, but it delivers a solid GVDG that is suitable as a basis for a subsequent Dijkstra query for finding a Voronoi path optimized with respect to path length and clearance.

3.2.3 Outside-to-inside propagation (OTI)

As described in Section 3.2.2 the ITO algorithm still has some problems with the chosen nodes V. The main problem with setting the nodes V to the Voronoi nodes VN is that the Voronoi nodes are highly dependent on the environment of a site s. Furthermore, the choice of the nodes is the basis for the GVDG and, in turn, for Voronoi paths. A suboptimal representation of a Voronoi cell by the Voronoi nodes, therefore, leads to suboptimal Voronoi paths and, in turn, to suboptimal disassembly paths. The nodes

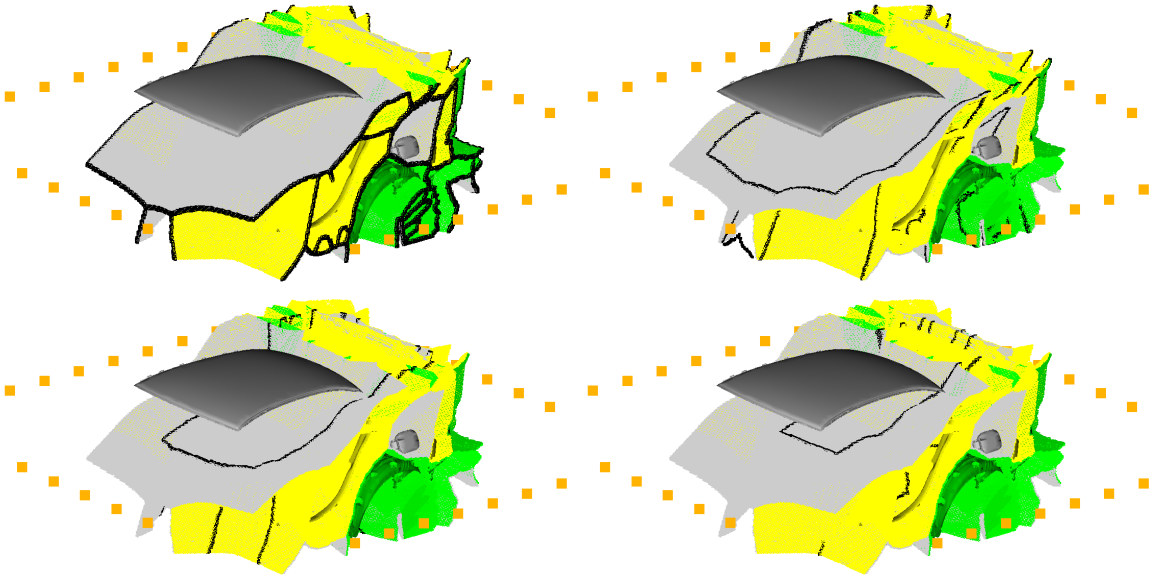

Fig. 27: The OTI propagation along the GVD. The black highlighted voxels represent the current wavefront.

should represent the Voronoi cell such that a subsequent Dijkstra query can determine the best Voronoi path to all body-in-white points (BIWPs). However, choosing the nodes that best fulfill this requirement is a hard task.

We therefore introduce the *outer-to-inner propagation (OTI)*, which works in the opposite direction. See Figure 27 for a visual representation of the propagation process. Instead of defining the start nodes from the Voronoi cells and propagating to the BIWPs, the OTI starts with the BIWPs and propagates to the inside of the GVD. The BIWPs are not part of the GVD. We define that each voxel from the GVD that is positioned at the boundary of the GVD (all black voxels in the first representation in Figure 27) is neighbored with its nearest BIWP. When the BIWPs start its propagation, therefore, the propagation jumps from the BIWP to the GVD. If a node propagates to a Voronoi cell and there is still no edge between the corresponding BIWP and the Voronoi cell, the edge is created and the voxel is added to the nodes V of the GVDG. This procedure has the advantage that the shortest Voronoi path between the BIWP and all Voronoi cells is set automatically because the metric respecting propagation process starts at the BIWPs and so inevitably hits the nearest voxel of a Voronoi cell first.

The OTI algorithm is described below. The GVDG propagation algorithm from Section 3.2.1 starts with all BIWPs as $startingVoxels$ which propagates as described

Algorithm 8 bool existEdge($voxel, neighbor$)

1: $vc \leftarrow VC(neighbor.site)$
2: **if** $\exists v \in vc\ ((v, voxel.startNodeGVDG) \in E)$ **then**
3: **return true**
4: **return false**

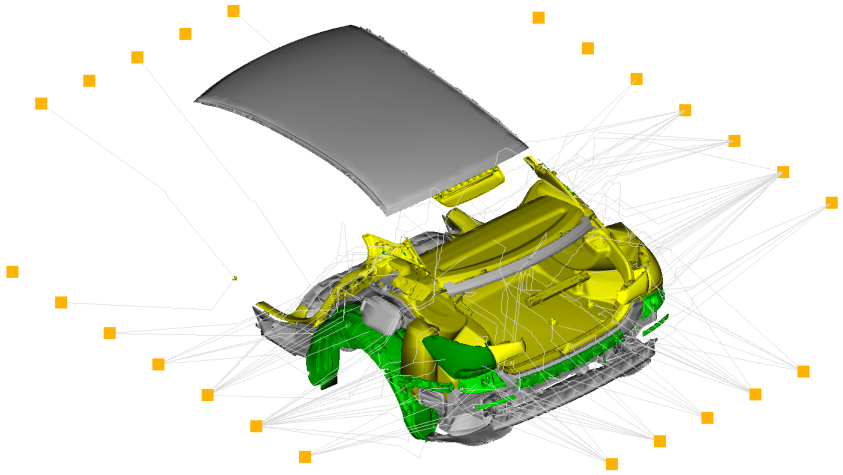

Fig. 28: The GVDG calculated with the OTI algorithm.

in Section 3.2.1 along the GVD. The *calculateGVDG* and *propagateVoxelGVDG* functions are unchanged. The *existEdge* function, which is called in Algorithm 6, is different to the function from the ITO algorithm and is therefore shown in Algorithm 8. In line 1 it sets the corresponding Voronoi cell vc to which *neighbor* belongs. To prevent flooding with very similar edges, the algorithm permits only one edge from one Voronoi cell to the same BIWP (lines 2, 3). If the edge is created in Algorithm 6 line 7 the *createEdge* function also adds the voxel n to the nodes V.

The resulting GVDG in this approach has a different form than the ITO GVDG. The structure of the ITO GVDG has many nodes with a lot of short edges. A Dijkstra query, therefore, navigates in short steps through a highly connected graph from node to node and ends at a BIWP. The OTI GVDG has only edges that connect a node on a Voronoi cell with a BIWP. The OTI GVDG therefore consists only of straight, non-zig-zagging Voronoi paths. For a detailed evaluation of the approaches presented, see Section 3.2.4. This circumstance results in short VPs which also means that a Dijkstra query is not necessary.

Table 6: The properties of the GVDGs for the three different approaches: straightforward approach (SF), inside to outside (ITO) and outside to inside (OTI). Tested on the data set (137 assembled parts) which is shown in Figure 29.

criterion	GVDG approach		
	SF	ITO	OTI
parts with an VP	67	74	94
average VP path length	1400	1444	1279

3.2.4 Evaluation

In this subsection, we evaluate the three different GVDG approaches presented: straightforward approach, ITO and OTI. The evaluation includes the number of isolated sites and average VP length. The three GVDGs created with the straightforward approach, ITO and OTI are shown in Figure 29. Our tests which are shown in Table 6 verify the theoretical considerations. The straightforward approach (SF) has the most isolated sites. The ITO approach also takes into account Voronoi faces and the medial axes and so, is more flexible and has fewer isolated sites (see Section 3.2.5 for a discussion why still some false isolated sites can occur). The OTI approach has not only the fewest isolated sites but also the shortest VPs.

3.2.5 Handle false isolated sites (HFIS)

As mentioned in Section 3.2, due to the crucial parameter δ_{MA} it is still possible that false isolated sites occur. See Figure 30 for an example. The medial axis (dotted line) is prevented for $d < \delta_{MA}$. Unfortunately, due to the lack of a medial axis, the GVDG is not connected and so isolates the screw even though the screw can be easily disassembled. The following propagation algorithm handles all cases of false isolated sites, i.e. it 'connects' these sites with the GVDG and subsequently finds a Voronoi path. We will not describe the *handleFalseIsolatedSites (HFIS)* algorithm in detail because it is merely a variant of the presented propagation GVD Algorithm 1 and GVDG Algorithm 4. We will therefore only describe the high-level procedure. The HFIS function is called after the ITO/OTI propagation algorithm from Section 3.2.1. First, we determine all the parts that are isolated, e.g. a_1 in Figure 31 a). Second, we

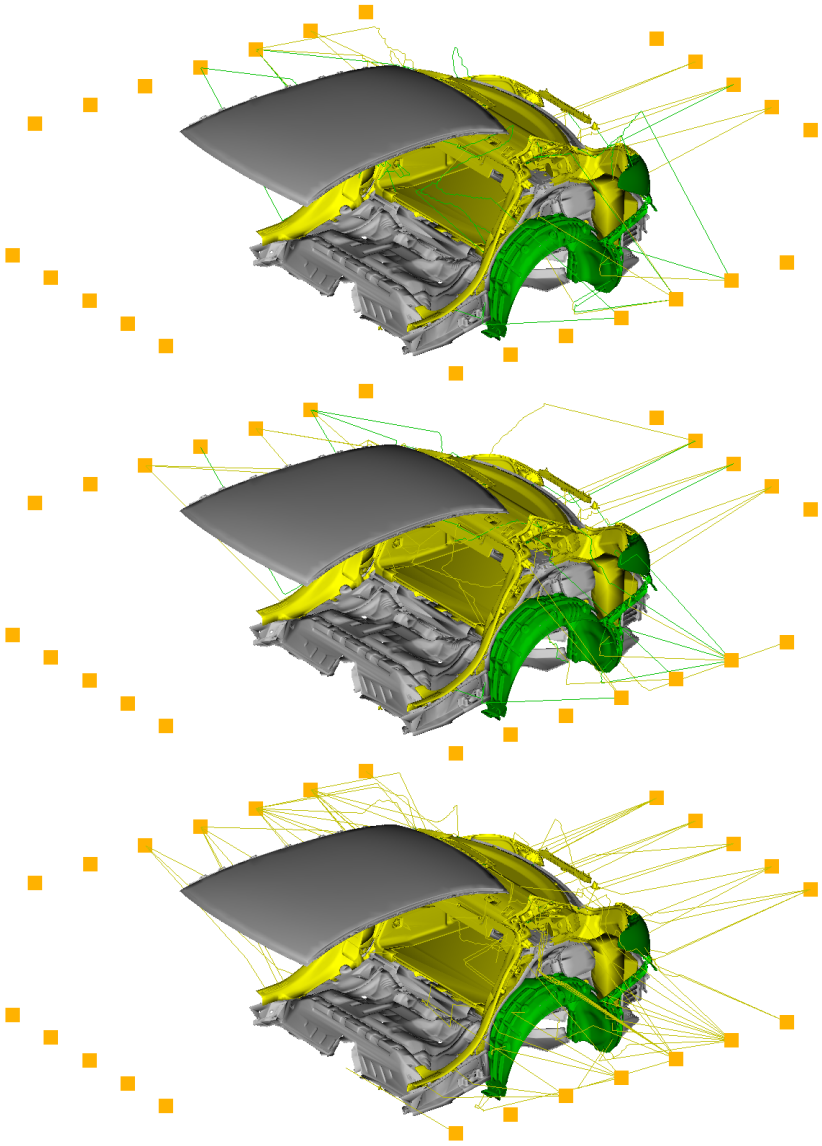

Fig. 29: Comparison of the resulting GVDGs: top - straightforward approach, mid - ITO and bottom - OTI.

delete the corresponding Voronoi cell $VC(a_1)$ b). Then we start a GVD propagation algorithm c), d). The starting nodes are are the voxels that represent the part (as shown in c)). If a voxel propagates to another Voronoi cell VC e) the voxel from the Voronoi cell VC is now a *connector*, i.e. a_1 is connected to the voxel. In other words, the red voxels in f) are now also part of the Voronoi cell of a_1. Subsequently, we start the GVDG propagation algorithm from Algorithm 4 again, only this case each connector also represents each connected part. In detail, we start the OTI algorithm with the BIWP and propagate to the inside of the GVD. If the propagation algorithm

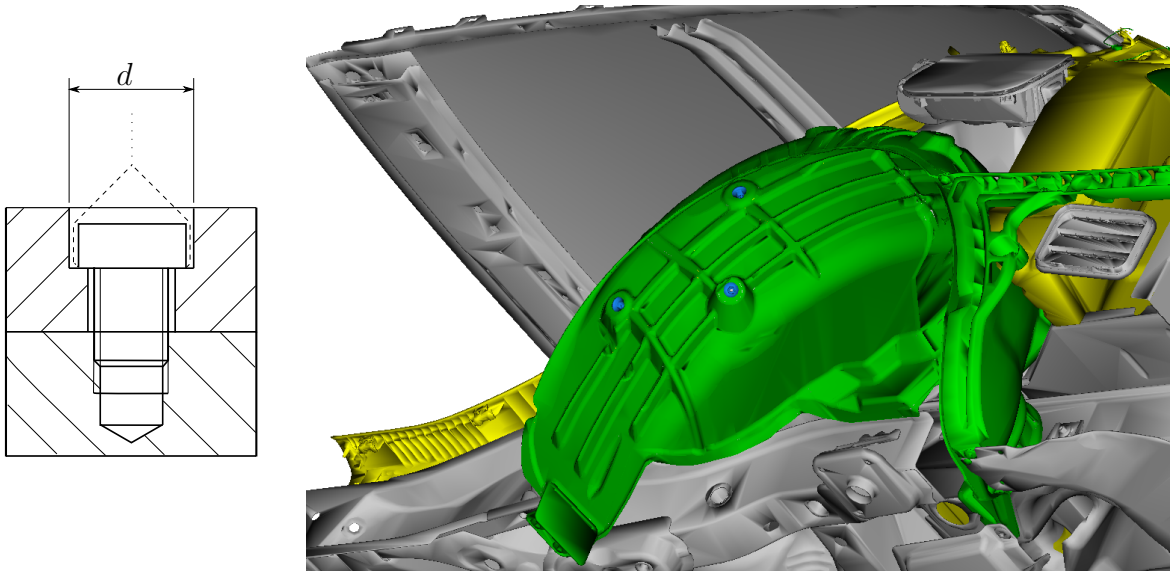

Fig. 30: Left: A screw that fastens the two different hatched parts. The dashed line represents the GVD between the screw and the hatched part. The dotted line represents the medial axis of the hatched part. Right: Some isolated nuts (highlighted in blue) in our real-world data set.

now reaches one of the connector voxels (red voxels in f)) it detects that a_1 is connected with these voxels and creates an edge in the GVDG from the BIWP to the connector voxel. It is not always certain that the isolated part can reach the connector voxel.

In addition to the 94 parts that have received a VP through the OTI approach, 37 others have been given a VP by the HFIS function. The remaining 6 parts from the 137 assembled parts are now truly isolated. This means that it is guaranteed that there is no feasible disassembly path in the current assembly state.

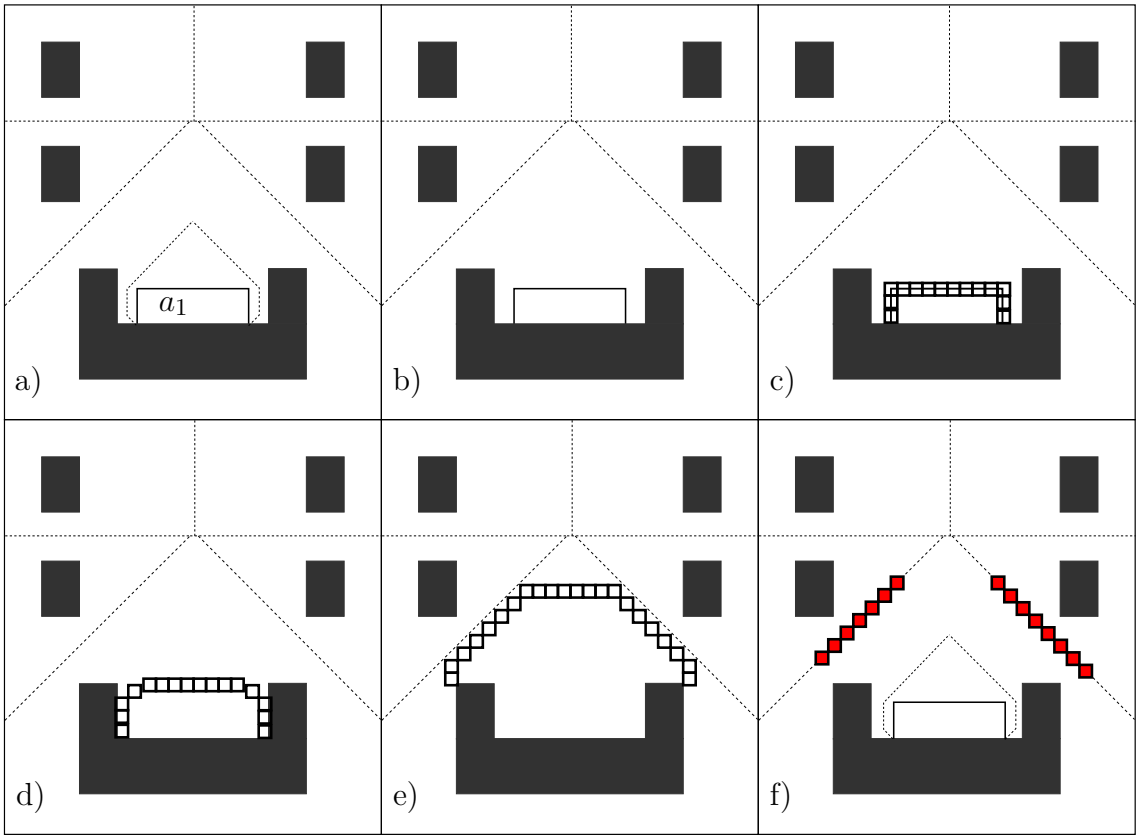

Fig. 31: Our *handleFalseIsolatedSites* algorithm shown in a simple two-dimensional example. The black-highlighted parts and a_1 are assembled parts. The dashed lines represent the GVD. The red-highlighted voxels are the connectors for a_1.

4

MOTION PLANNING

In this chapter, we will focus on motion planning and how we use it for assembly sequence planning. After introducing the motion planning problem, we provide an overview of the specific requirements in the context of ASP. To accommodate these requirements, we subdivide the motion planning task into two disjoint phases: the *NEAR-* and *FAR planning phases* (Masan, 2015). The NEAR planning phase describes the unlocking of fastened parts. The FAR planning phase describes the navigation from an unlocked position to a goal position. A special feature of the unlocking process is that the part to be disassembled may initially collide with its surroundings which fundamentally contradicts the definition of a feasible disassembly path. However, we broaden the original definition in this case and discuss state-of-the-art motion planners that are designed to still find a reasonable disassembly path for the NEAR planning phase. An appropriate motion planner for the FAR planning phase should quickly find a short disassembly path to one of the goal positions.

Next, we present the main contribution of this chapter: our novel *Expansive Voronoi Tree (EVT)* motion planner, an optimized motion planner for the FAR planning phase in the context of ASP. The EVT samples along a VP, which is extracted from the GVDG. Our experiments show that the EVT finds short disassembly paths faster than state-of-the-art motion planners. The ability to find short disassembly paths more quickly comes with the high runtime costs of the GVDG calculation. However, the runtime calculation of the GVDG is distributed across all assembled parts and so negligible. In summary, in the context of ASP, the EVT is a better motion planner than state-of-the-art algorithms in terms of runtime and path quality.

4.1 Motion planning problem

The *motion planning problem*, also known as the *path planning problem*, concerns the process of finding a connected sequence of configurations, a so called *disassembly path*, for a *robot* to be moved. The disassembly path must navigate the robot from the starting point to a goal position such that the robot does not collide with a given set of static *obstacles*. Next, we define the problem formally.

Definition 8 (*Configuration space and workspace*)

Given a mesh m. Then the configuration space

$$C := \left\{ A | A = \begin{pmatrix} R & t \\ 0_{1\times 3} & 1 \end{pmatrix} \in \mathbb{R}^{4\times 4}, R \in \mathbb{R}^{3\times 3}, t \in \mathbb{R}^3, R^T R = R R^T = I, det(R) = 1 \right\},$$

describes all motions of m in the space $\mathbb{R}^3$. With c_t we indicate the translation t and with c_R the rotation R of a configuration $c \in C$. We indicate the mesh m' that results after applying a configuration c to m as $c(m)$.
Given a set of obstacles O, where each obstacle $o \in O$ is represented via a triangulated mesh. We then define the set of collision-free configurations $C_{free} \subset C$ as the set of configurations for which $c \in C_{free}$ applied to m has no collision and is not fully enclosed by any of the obstacles $o \in O$. The set of configurations that leads to a collision or an enclosure is defined as $C_{obst} := C \setminus C_{free}$. We define the start configuration of m as c_{init} and assume $c_{init} \in C_{free}$.
In addition, with the workspace $\subset \mathbb{R}^3$, we define the space in which the meshes are positioned.

Definition 9 (*Disassembly path I*)

Given a triangulated mesh m with its initial configuration $c_{init} \in C_{free}$, a set of obstacles O and a set of possible goal points $B \subset \mathbb{R}^3$. Then a feasible disassembly path

$$dp := \left\{ c_{init}, c_1, ..., c_n, c_{goal} | c_{init}, c_{goal}, c_i \in C_{free} \forall 1 \leq i \leq n, \exists b \in B \ (c_{init_t} + c_{goal_t} = b) \right\}$$

is a discrete sequence of configurations such that m is navigated from its initial configuration to one of the goal points B. The distance between two following configurations of dp is sufficiently small.

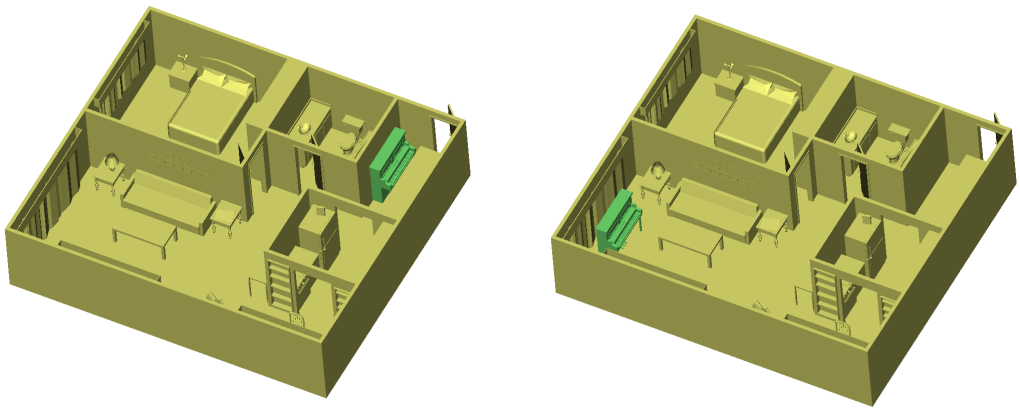

Fig. 32: The piano in its initial (left) and goal position (right) (Schneider, 2017).

Note: The term 'sufficiently small' depends on the size of the robot and the scene.

Definition 10 (*Motion planning problem I*)

Given a dynamic object m, also called robot, with its initial configuration $c_{init} \in C_{free}$, a set of obstacles O and a set of possible goal points $B \subset \mathbb{R}^3$. The motion planning problem here is to find a feasible disassembly path dp for m.

In our definition, we generalized the motion planning to a set of possible goal points B.

Figure 32 shows the famous '*piano movers problem*'. In this example, a search is performed for a disassembly path so that the piano moves from the initial position to the goal position such that the piano does not collide with its environment (highlighted in yellow).

The common approach to solving complex motion planning problems - i.e. feasible disassembly paths consisting of multiple translations and rotations, is done with so called *sampling-based motion planners*. Those algorithms use an underlying data structure for the configurations, start with the given start configuration $c_{init} \in C_{free}$ and try with samples to extend this data structure. The aim is to describe the free space C_{free} with the configurations in the data structure and to find, with a sufficient number of configurations within it, a feasible disassembly path to one of the goal positions B. We subdivide the sampling-based motion planners into three different

categories - *graph-*, *tree-* and *GVD*-based motion planners and will now provide an overview of related work.

4.1.1 Related Work

Motion planning is an extremely broad field with extensive research. For a detailed overview of motion planning, we refer to the survey from Yang *et al.* (2016). There are reams of works that tackle different requirements such as navigating narrow passages in the configuration space (Hsu *et al.*, 1997), maximizing the clearance (Denny *et al.*, 2014) or finding optimized paths (Karaman and Frazzoli, 2011). We will now provide an overview of the general state-of-the-art techniques.

Graph-based:

One pioneering work for sampling-based motion planners is the 'Probabilistic Roadmap' (PRM) (Kavraki *et al.*, 1996). This is a graph-based algorithm that consists of a construction and query phase. In the construction phase, the roadmap (graph) is created. The algorithm samples random configurations and connects them with the neighbored configurations. In the query phase, once the start and goal configurations have been successfully connected to the graph, a shortest-path algorithm delivers the path with an optimized condition such as path length or clearance (Wilmarth *et al.*, 1999), (Holleman and Kavraki, 2000). The process of creating the roadmap is extremely time-consuming, however, and varies from one robot-obstacle configuration to the next, which means that it is not suitable in the context of ASP.

Tree-based:

Algorithm 9 bool findPath($robot, obstacles, c_{init}, B, maxTime$)

1: $configTree.insert(c_{init})$
2: **while** elapsedTime $< maxTime$ **do**
3: $\quad c_{rand} \leftarrow randomConfig$
4: $\quad configTree.extend(c_{rand})$
5: $\quad$ **if** $goalPointReached(robot, c_{rand}, B)$ **then**
6: $\quad\quad$ **return true**
7: **return false**

Tree-based motion planners generally involve the following procedure which, is shown in Algorithm 9. They set c_{init} as the root of the tree (line 1). Next, they sample

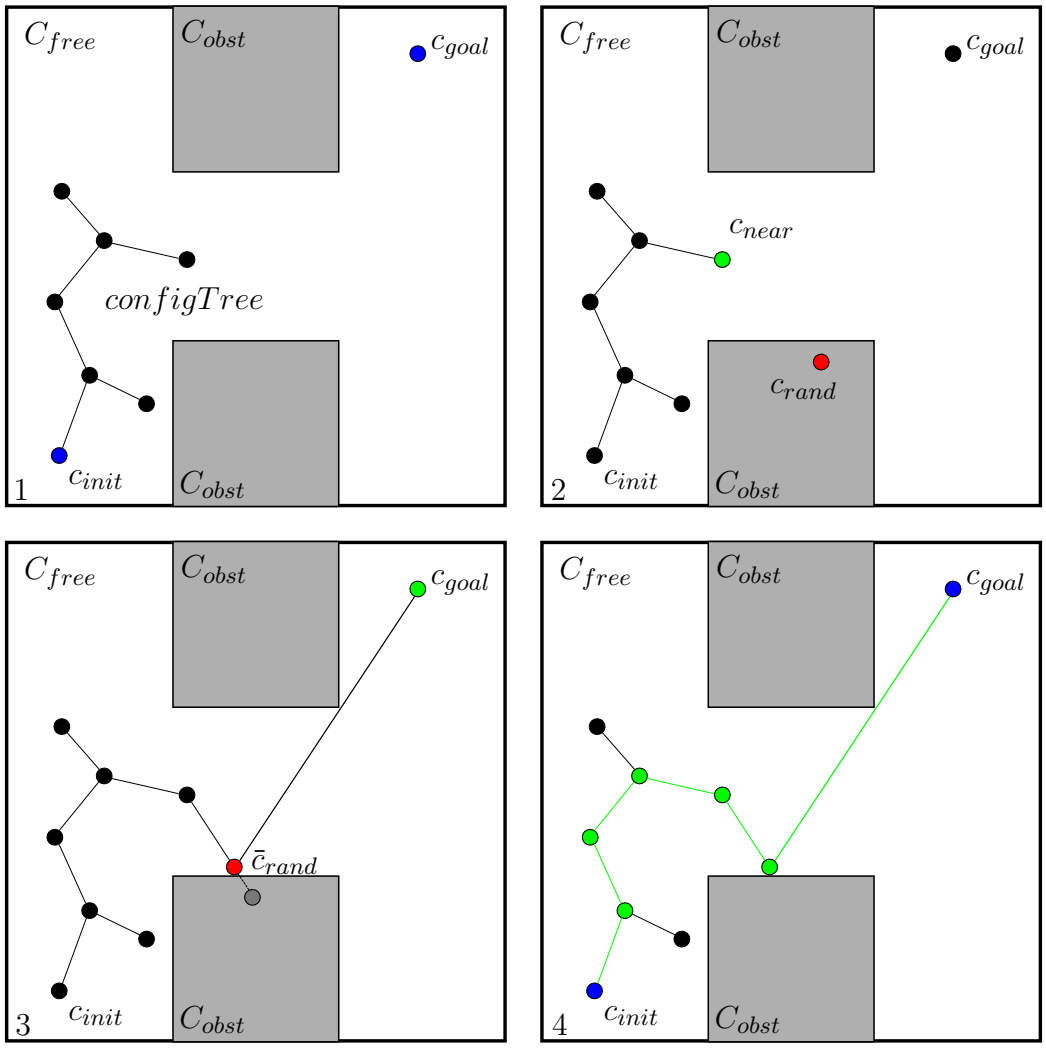

Fig. 33: The steps from Algorithm 9 lines 3 - 6 for the ordinary RRT algorithm.

and create a new configuration and connect it with the tree (line 3, 4). The while loop terminates when the algorithm finds a feasible disassembly path (line 5, 6) or the maximum time limit is reached (line 2). The biggest difference among various motion planners lies in how they sample and create new configurations. Next, we describe some common sampling-based approaches.

The most widely used approach with many variations is the 'Rapidly-Exploring Random Tree' (RRT) from Lavalle (1998). See Figure 33 for a visualization of the steps which are described below. The RRT uses the starting configuration as its root. It then uniformly samples a random configuration c_{rand} in the configuration space and attempts to connect c_{rand} with the nearest configuration c_{near} of $configTree$ (Figure 33, 2). For $c_{rand} \in C_{obst}$ the RRT adds the furthest possible configuration $\bar{c}_{rand} \in C_{free}$ to the

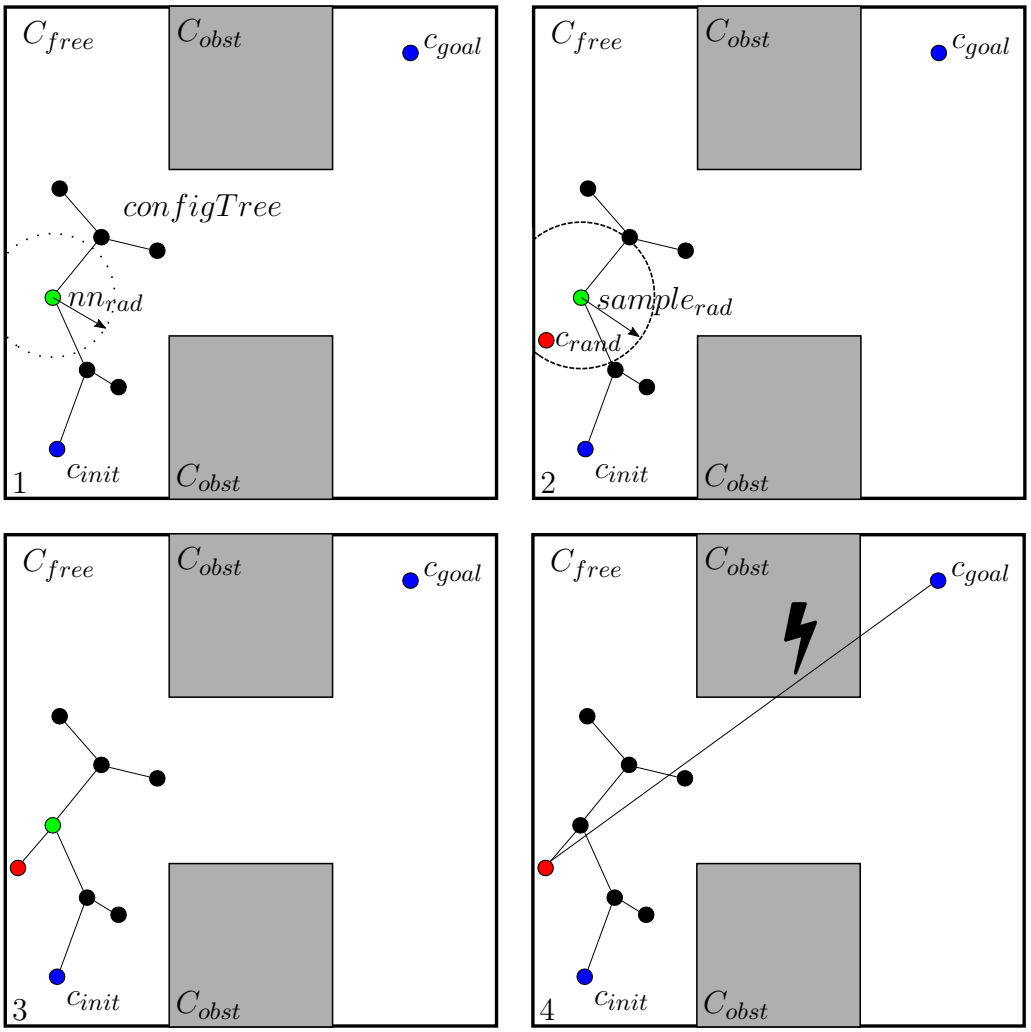

Fig. 34: The steps from Algorithm 9 lines 3 - 6 for the ordinary EST algorithm.

$configTree$ and discards c_{rand} (Figure 33, 3). The connecting process of c_{rand} works as follows: Start at c_{near} and gradually approach c_{rand}. Add the feasible configurations from each step to the tree. Stop, if a configuration is no longer feasible. The last feasible configuration is $\bar{c}_rand$. After each sampling step, the RRT attempts to connect the last configuration with the goal position c_{goal}. The advantage of this sampling strategy is that the RRT explores the entire space very effectively. It terminates as soon as a feasible path for the robot is found. In its pure form, however, the uniform sampling approach means that it does not work well for narrow passages and does not return an optimized path.

The extension RRT* (Karaman and Frazzoli, 2011) returns a path that optimizes a user-defined criterion such as path length. The RRT* does not terminate as soon as it finds

a feasible path but attempts to optimize the path by further sampling configurations until a given time limit is reached. After each sampling step, the algorithm rewires the configurations in the neighborhood of the sampled configuration, attempting to optimize the criterion. The RRT* algorithm is much slower than the RRT because rewiring the tree is a time-consuming task. Since the sampling strategies are identical, the RRT* has the same problems with narrow passages as the RRT.

The 'Expansive Spaces Tree' (EST) (Hsu *et al.*, 1997) is a tree-based planner with a sampling strategy that is optimized for narrow passages. The EST sampling behavior is shown in Figure 34. The EST extends the configuration with the fewest neighbors within the radius nn_{rad}. Around this configuration, the EST samples within the radius $sample_{rad}$ a new configuration c_{rand} and attempts to connect the sampled configuration with the configuration around which sampling was performed. If the sampled configuration is in the obstacle, i.e. $c_{rand} \in C_{obst}$, the EST just like the RRT adds the furthest possible configuration to the $configTree$ and discards c_{rand} (see Figure 33, 3). The attempt to connect the last configuration of the $configTree$ with c_{goal} is shown in Figure 34, 4. In this example, the connection shown is not feasible and the EST continues with the while loop. The fact that the EST extends the configuration with the fewest neighbors means that this strategy focuses on exploring narrow passages. However, this comes to the cost that the EST needs more time for a coarse exploration of the complete configuration space as a whole.

For tree-based planners with a specified goal position, the 'connect' version (Kuffner and LaValle, 2000) can always be used. The 'connect' version expands an additional tree at the goal configuration and attempts repeatedly to connect both trees.

GVD-based:

A number of works address the GVD (medial axis) in the configuration space (Wilmarth *et al.*, 1999), (Denny *et al.*, 2014). These planners attempt to find a path with maximal clearance in the configuration space. They optimize the clearance of each configuration, which makes these motion planners even more time-intensive than the aforementioned tree-based planners.

Most approaches that use the GVD in the workspace take into account only point robots (Seda, 2007), (Seda and Pich, 2008), (Wang *et al.*, 2011) for which no rotation needs to be calculated. General objects are studied by Hoff *et al.* (2000). The robot's center is placed on the path VP derived from the GVD and pushed forward. The robot is translated and rotated according to geometry and position within the distance field.

However, calculating the distance field forces on the robot for every motion planning step the time-consuming evaluation of every point of the robot within the distance field.

Foskey *et al.* (2001) presents a hybrid GVD- and EST-based motion planner. The robot's center is placed on the VP and pushed forward. The robot's rotation is deduced from the calculation of a difference quotient. All positions on the VP are marked as unfeasible *bridges* where the robot is in collision with an obstacle. The algorithm then attempts to connect these bridges with an EST-connect algorithm. But, there are cases where every position on the VP is in collision, meaning that the entire path needs to be bridged. In these cases, the hybrid approach in Foskey *et al.* (2001) is simply the EST-connect. Our GVD-based planner, which uses the VP for the EST-sampling itself, is the more general approach and has a faster reliable runtime.

Motion planning in the context of ASP with real-world CAD data entails a range of special challenges. As mentioned in Section 2.1.3 real-world CAD data consists of flexible parts that have an initial collision *near* its installed position. In reality, they can be 'unfastened' only if they are deformed. In addition, *far* from the installed position, the large workspace for complex real-world scenarios permits many different feasible disassembly paths for each assembled part. These different disassembly paths differ in terms of length and clearance. An ASP framework should therefore also be capable of determining optimized disassembly paths. To address both these different requirements, we subdivide the motion planning process into two disjoint phases: the NEAR- and FAR planning phase (Masan, 2015). Roughly speaking, the NEAR planning phase describes the 'unlocking' of an assembled part, while the FAR planning phase describes the 'navigating' to a nearby goal position.

4.2 NEAR planning phase

The NEAR planning phase describes the 'unlocking' of an assembled part. For our test data, we define that the NEAR planning phase ends when the assembled part has traveled at least 50 mm from c_{init} and is not in collision. As mentioned in Section 2.1.3, assembled parts need to pass narrow passages in their immediate environment. It is possible that subparts are in an initially collision and need to be deformed during the disassembly process. Simulating the exact deformation is a very challenging task. The algorithm needs to know which parts are deformable and how exactly they behave when

subjected to force. If the material properties are known, this can be simulated with 'finite elements method' (FEM). More precisely, a physical simulation is needed for accuracy. Our test data, however, consists only of meshes, but even for small examples, the FEM simulation process takes a few hours and so is not suitable in the context of ASP where hundreds of parts need to be disassembled. We will therefore look at pure geometrical algorithms that approximate the physical simulation. The following describes the existing algorithms that find a disassembly path for the NEAR planning phase for components with flexible parts. They are not as accurate as FEM but a lot faster. In the NEAR planning phase, we do not assume that $c_{init} \in C_{free}$ or even the configurations of the disassembly path can be part of C_{obst}. Nevertheless, the existing works presented here still ensure that the disassembly path is plausible.

4.2.1 Related Work

Schneider (2017) presents a motion planner that finds feasible disassembly paths for parts that have flexible (sub)parts. The motion planner first detects flexible (sub)parts using *Position Based Dynamics* (Müller *et al.*, 2007) and, for these (sub)parts, in subsequent motion planning a specific local intersection volume. The computed disassembly paths can consist of arbitrary translations and rotations. However, the algorithm requires for each part an individual parameter that specifies the permitted local intersection volume.

The *Iterative Mesh Modification Planner (IMMP)* from Hegewald *et al.* (2022) is an alternating method of controlled mesh deformations and planning attempts that are performed until a maximum number of iterations is reached or a path is found. With the mesh deformations, the method is capable of eliminating any unavoidable collisions caused by flexible fastening elements and overpressure of components. For the deformation of fastening elements, the method relies on the computation of a saliency map (Hegewald *et al.*, 2021) of the robot that extracts the fastening elements as salient regions. By eliminating any unavoidable collisions, the method is capable of applying the conventional *Expansive Spaces Tree (EST)* (Hsu *et al.*, 1997) motion planner to compute a collision-free disassembly path. The delivered disassembly paths are reasonable and can be computed very quickly compared with exact simulations.

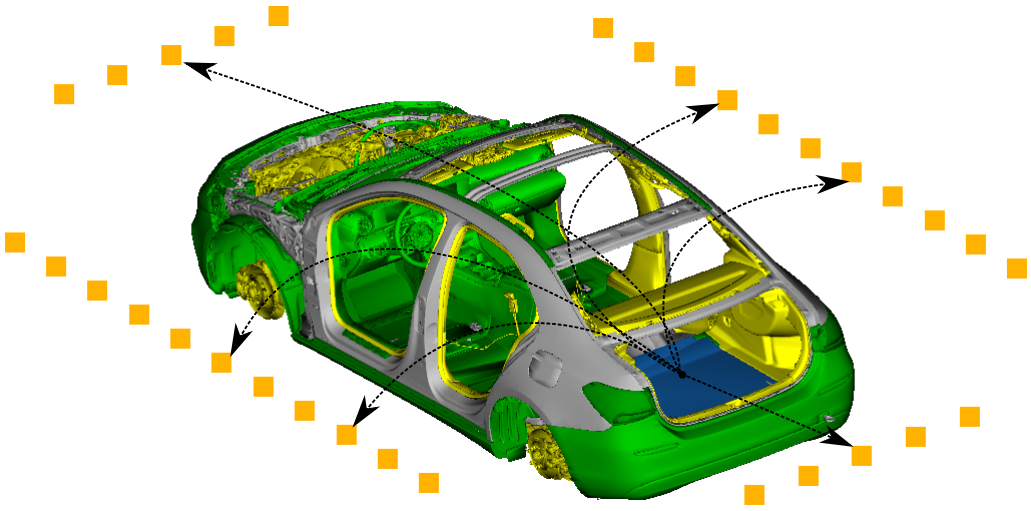

Fig. 35: The dashed arrows show possible feasible disassembly paths for the blue-highlighted floor plate in the trunk.

4.3 FAR planning phase

The FAR planning phase describes the 'navigating' of an assembled part from the end of the NEAR planning phase to a goal position B. We assume therefore that the part to be disassembled was successfully disassembled during the NEAR planning phase and so it has a collision-free initial configuration $c_{init} \in C_{free}$ at the beginning of the FAR planning phase. We discuss the requirements that motion planners must fulfill for the FAR planning phase in the context of ASP.

As shown in Figure 35, complex real-world scenarios are characterized by a large installation space that with numerous assembled parts and for many of them, feasible disassembly paths to multiple goal positions. Furthermore, due to a constricted installation space and the geometry of the part to be disassembled, the path may contain narrow passages in between. On one hand, therefore, a motion planner must explore the entire scene globally, especially if certain criteria such as path length and clearance are to be optimized: on the other hand, it must overcome narrow passages locally.

Furthermore, a motion planner with a fast running time is crucial for efficient ASP. The most obvious reason is that a high number of assembled parts leads to multiple motion planning attempts. The other reason is the following: An ASP algorithm always needs to decide for which assembled part the next motion planning query will be started.

Due to the complex design of the assembly, the algorithm will inevitably often choose an assembled part that cannot be disassembled in the current state because other assembled parts need to be disassembled first. The usual procedure is that the motion planner attempts to find a feasible disassembly path until a given time limit t_{max} is reached, whereby, an ideal value for t_{max} must be as small as possible to allow a good running time for these *impossible disassembly queries (IDQ)* but large enough to ensure that for *every single possible query* the motion planner still finds a disassembly path with a high probability p. We can say, that a motion planner has a reliable running time $t(p)$ if the planner for each part that can be disassembled finds a path with a probability of at least p within the time bound $t(p)$.

Summarized, the following requirements are placed on a motion planner in the context of ASP for the FAR range planning phase:

(i) Explore the complete scene coarsely but also overcome narrow passages,

(ii) find a path that is short and ideally has enough clearance,

(iii) find the path within a short reliable runtime $t(p)$ for a high pathfinding probability p.

We show that our novel Expansive Voronoi Tree (EVT) motion planner fulfills all the aforementioned requirements for motion planning in the context of ASP. It also works in the configuration space but its translational sampling is based on the approximated GVD from Chapter 3 in the workspace. The property of the GVD is that it delivers disassembly paths with maximal clearance for point robots. This is not generally true for arbitrary robots, but we extract an estimated disassembly path (VP) from the GVDG which is calculated with the OTI propagation method from Section 3.2.3. Our approach is now as follows: We look for a valid disassembly path close to the VP. A search is performed for the translational part near the VP and the rotational part is randomly sampled.

One advantage of GVD-based planning for a subsequent ASP is that the GVD calculated once for one assembly situation is the same for all assembled parts. The runtime bottleneck of GVD-based path planning, the calculation of the GVD, is therefore distributed across all assembled parts to be disassembled in the current assembly state.

4.4 NEAR- and FAR planning phase

As mentioned in Section 4.2, the ability to handle real-world data requires broadening of the original definition of a disassembly path such that the 'controlled' collision of fastening elements is still feasible in the highly constricted environment of the assembled part. We thus broaden our definitions so that they also apply to real-world data.

Definition 11 (*Disassembly path II*)

Given a triangulated mesh m with its initial configuration $c_{init} \in C$, a set of obstacles O and a set of possible goal points $B \subset \mathbb{R}^3$. A feasible disassembly path dp is therefore the union of the disassembly path for the NEAR- and FAR planning phase

$$dp := dp_{near} \cup dp_{far}$$

where we allow for the NEAR planning phase configurations out of C_{obst}

$$dp_{near} := \{c_{init}, c_1, ..., c_n, c_{near_{end}} | \, \|c_{init} - c_{near_{end}}\|_2 \leq d, c_{near_{end}} \in C_{free}\}$$

and the FAR planning stays the original defintion

$$dp_{far} := \left\{c_{near_{end}}, c_1, ..., c_n, c_{goal} | \exists b \in B \; (c_{init_t} + c_{goal_t} = b)\right\} \subset C_{free}.$$

The distance between two following configurations of dp is considered to be sufficiently small. Furthermore, the amount of permitted collision during the NEAR planning phase must be controlled and reasonable, i.e. the permitted collision should contribute only to the plausible disassembly of flexible fasteners.

Note: The term 'sufficiently small' depends on the size of the robot and the scene.

Definition 12 (*Motion planning problem II*)

Given a dynamic object m, also called robot, with its initial configuration c_{init}, a set of obstacles O and a set of possible goal points $B \subset \mathbb{R}^3$. The motion planning problem is therefore to find a feasible disassembly path dp according to Definition 11 for m.

4.5 Expansive Voronoi Tree

In the following, we present our new EVT motion planning algorithm. First, we use the GVDG from Section 3.2.3 and search for a VP. Next, we sample along this Voronoi path. The basis of our sampling strategy is the EST motion planner which is optimized for narrow passages. We change the translational sampling of the classical EST such that it adapts to the clearance situation and goes in the forward direction of the Voronoi path VP. In addition, the Voronoi path acts as a magnet that attracts the configurations back to itself.

We will now provide a detailed description of our algorithm, also taking into account the pseudocode. For the chosen parameters we refer to the experiments in Section 4.6.

Algorithm 10 bool EVTfindPath(*robot*, *obstacles*, VP)

```
1: setSamplingAndSearchRadii(robot.size)
2: goalAchieved ← false
3: while !goalAchieved ∧ !timeLimitReached do
4:     goalAchieved ← expandTree(VP)
5: return goalAchieved
```

EVTfindPath is our main method, and it receives the robot, obstacles and a Voronoi path VP as input. The VP was extracted with the OTI approach presented in Section 3.2.3. In line 1, we set - as a function of depending on the size of the robot - the nearest neighbor search radius $nnRad$, translational sampling radius $transRad$ and rotational sampling radius $rotRad$. We define the size of a robot as the length of the diagonal of the OBB. Once the $goalAchieved$ condition initial is set to false (line 2), our tree-expanding main loop is executed (lines 3 - 4). This loop terminates when the goal of VP is achieved or the given time limit is reached.

Algorithm 11 bool ExpandTree(VP)

```
1: c ← getConfigWithLeastNeighbors(nnRad)
2: c̄ ← sampleNextConfig(c, VP)
3: connectSuccessful ← connectConfigs(c, c̄)
4: if connectSuccessful ∧ c̄.v == VP.last() then
5:     return true
6: return false
```

ExpandTree attempts to expand the EVT along the Voronoi path VP in each step. First (line 1), we choose from the EVT the configuration c that has the fewest neighbor

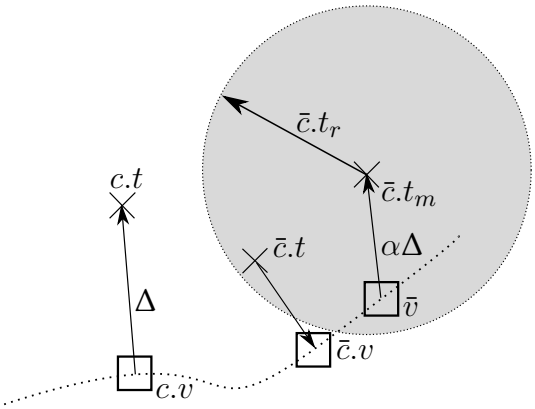

Fig. 36: A schematic presentation of the translational sampling. The dotted line represents the VP. The shift Δ defines the distance vector from $c.t$ to its nearest voxel $c.v$. The next translation will be sampled within a sphere with center $\bar{c}.t_m := \bar{v} + \alpha\Delta$ and radius $\bar{c}.t_r$. We retract new samples with the factor $0 < \alpha < 1$ back to the VP.

configurations within the search radius *nnRad*. This is similar to the procedure in the EST algorithm. Next, based on c, we sample a new configuration $\bar{c}$ (line 2) in **SampleNextConfig**. We attempt to connect $\bar{c}$ with c (line 3). If $\bar{c}$ has the goal position as the nearest voxel on VP, this means that a feasible disassembly path was found and we return true (lines 4 - 5). Otherwise, false is returned (line 6).

Algorithm 12 Configuration **SampleNextConfig**(c, VP)

1: $\bar{c} \leftarrow$ new Configuration()
2: $\bar{v} \leftarrow$ getReferenceVoxel$(VP, c.v, robot.size)$
3: $\bar{c}.t \leftarrow$ sampleTranslation$(c.t, transRad, \bar{v})$
4: $\bar{c}.r \leftarrow$ sampleRotation$(c.r, rotRad, \bar{v})$
5: $\bar{v}.samplings++$
6: $\bar{c}.v \leftarrow$ getNearestVoxel$(VP, \bar{c}.t)$
7: **return** $\bar{c}$

SampleNextConfig samples a new configuration $\bar{c}$ with respect to the configuration c in the forward direction of VP.

Line 1: Our algorithm instantiates the new configuration $\bar{c}$.

Line 2: We search for the so called *reference voxel* $\bar{v} \in VP$ (see Figure 36 for a graphical representation): The nearest voxel of c is $c.v$. As reference voxel $\bar{v}$ of c we choose the first voxel in forward direction of VP for which $s(c.v, \bar{v}) > robot.size/4$ holds. Where $s(v_i, v_j) := \sum_{k=i}^{j-1} d(v_i, v_{i+1})$ describes the path length from v_i to v_j. By sampling in forward direction of VP we can quickly pass by passages with ample clearance. In

wide open regions the algorithm instantly finds a feasible configuration in the sampling step described below and moves forward with one sampling step $robot.size/4$.

We will now provide an intuitive description of why the EVT performs well in narrow passages even if we have the big step $robot.size/4$ in the translational sampling. The $connectConfigs$ function from Algorithm 11 in line 3 works like the aforementioned tree-based motion planners (see Figure 33 3) on page 54). This means that the tree is expanded in small steps from the original configuration c to $\bar{c}$ until a configuration is in collision (as shown in Figure 33 3) with $\bar{c}_{rand}$). Under the assumption that the forward direction of the VP is correct we still expand the tree in the right direction even if the new sampled configuration $\bar{c}$ is too far away and results in a collision.

Line 3: We sample the translation $\bar{c}.t$ of our new configuration $\bar{c}$, extending c. Refer again to Figure 36. The configuration c has a translational shift $\Delta := c.t - c.v$ to its nearest voxel $c.v$. The shift Δ indicates that it was most likely the case that no collision-free configuration was found that is closer to the Voronoi path VP than Δ. We set the center $\bar{c}.t_m$ of the sphere, in which we want to sample the translation $\bar{c}.t$, of the new configuration $\bar{c}$ as $\bar{c}.t_m := \bar{v} + \alpha\Delta$. We thus sample in the forward direction of VP, still taking into account the previous shift Δ. By multiplying the shift Δ with $\alpha \in (0, 1)$ we retract the new configuration back to the Voronoi path VP. The sample radius is defined as $\bar{c}.t_r := f(\bar{v}.sampling) \cdot transRad$.

We hereby define $f : \mathbb{N}_0 \to [0, 1]$ as a strictly increasing function. This function takes the number of sample attempts $\bar{v}.samplings$ around the voxel $\bar{v} \in VP$ as its argument. This number indicates how difficult it was for the EVT to find a feasible configuration near $\bar{v}$. The function $f(x) := \min\{x/50, 1\}$ worked well in our tests. By multiplying $f(\bar{v}.samplings)$ to the sample radius $transRad$, we ensure that the radius increases only up to $transRad$ if needed.

For a detailed explanation of how we determine a sampling point within a sphere read the following Section 4.5.1.

Line 4: Each rotation is represented as a point on the unit sphere. We sample $\bar{c}.r$ on the unit sphere around the point $c.r$ but consider only configurations that have at most the distance $f(\bar{v}.sampling) \cdot rotRad$ to $\bar{c}.r$. In other words, $distance(\bar{c}.r, c.r) \leq f(\bar{v}.sampling) \cdot rotRad$ where it is the distance on the surface on the unit sphere.

Line 5: Increase the counter $\bar{v}.samplings$ that counts how many times we sampled around the voxel $\bar{c}.v$.

Line 6: We set $\bar{c}.v$ to the nearest voxel of $\bar{c}$ on VP. For a large sampling radius this nearest voxel $\bar{c}.v$ is not necessarily equal to the reference voxel $\bar{v}$, refer again Figure 36.

4.5.1 Analysis

We theoretically evaluate how different distribution functions for our iterative translational sampling (**SampleNextConfig** line 3) affect the workspace-exploring property of our EVT.

For simplicity, let's assume that all voxels of the Voronoi path VP are arranged along a straight line. By sampling in each step within a sphere with a radius of at most $transRad$, e.g. with a uniform distribution, the maximum distance we can move away from $vPath$ in one sampling step is $transRad$. The algorithm maximizes the distance to $vPath$ by always sampling in the same direction orthogonally to $vPath$. The distance is therefore $d_1 = transRad$ in the first step, $d_2 = \alpha \cdot transRad + transRad$ in the second and

$$d_i = \sum_{k=0}^{i-1} transRad \cdot \alpha^k$$

for sampling step i. For $i \to \infty$ the sum converges from below toward $d = transRad/(1-\alpha)$. Thus, with sampling only inside the sphere the EVT algorithm is not capable of exploring the entire configuration space because the distance from the sampled configurations to the VP is bounded.

Therefore we sample Gauss distributed around the sampling center $\bar{c}.t_m$ with variance $transRad$. Since the Gauss distribution is not bounded by the surface of the sphere, the translational sampling has a small probability to explore the complete workspace. Most of the samples are nonetheless near VP, but the EVT can expand to the entire scene. It therefore explores the configuration space over the long term like a normal EST.

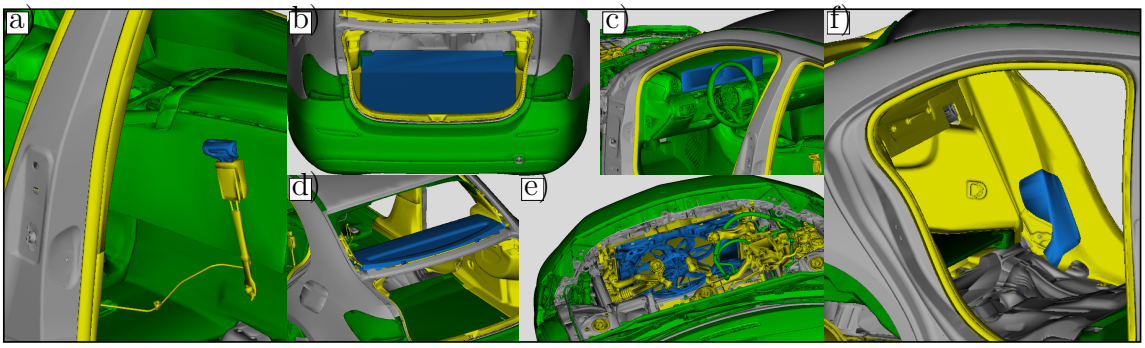

Fig. 37: The blue-highlighted parts show our test data: a) seat belt clip (3,275 triangles), b) trunk ground plate (8,429 triangles), c) display (40,176 triangles), d) hat rack (17,743 triangles), e) ventilator (70,036 triangles), f) covering plate (2,365 triangles).

4.6 Experiments

4.6.1 Implementation and test details

We test and compare our EVT against the following algorithms which we have reimplemented: RRT (Lavalle, 1998), RRT* (Karaman and Frazzoli, 2011), EST (Hsu *et al.*, 1997) and EST-connect (Kuffner and LaValle, 2000). For each part to be disassembled we executed 100 runs for every algorithm and each run for at most 120 seconds. For all delivered disassembly paths we used a path-smoothing algorithm with a runtime of 1 second. We performed these extensive tests for the six parts shown in Figure 37. We also compared our EVT against the RRT for all parts but for only one run for the data set which is shown in Figure 38.

The comparison does not include the aforementioned hybrid motion planner (Foskey *et al.*, 2001) because it is a pure EST-connect in situations where the center of the robot has to leave the Voronoi path. This is quite often the case, for example for the parts b), d), e) in Figure 37.

The RRT, RRT* and EST motion planners are implemented as multi-goal planners to all surrounding goal positions B, i.e. the motion planners attempt to connect the last collision-free configuration with all goal positions B that are closer than 1500 mm. Our EVT and the EST-connect algorithm attempt to find a disassembly path to the goal position of $vPath$. The EST-connect therefore grows a second tree starting from the goal position. Our parameter tests showed that the following sampling parameters are optimal for the EVT, EST and EST-connect: $transRad = 0.5 \cdot robot.size$, $rotRad = 0.1$

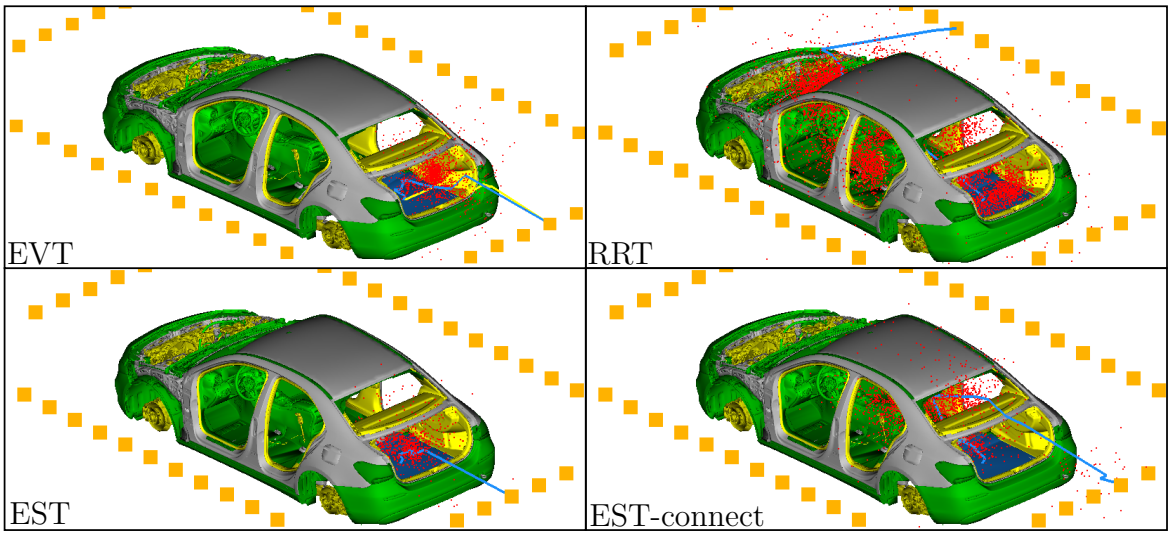

Fig. 38: The sampling behavior of the motion planners EVT, RRT/RRT*, EST and EST-connect for the assembled part in Figure 37 b): The red dots are translational sampling points, while the blue line shows the translational part of the computed disassembly path. The yellow line for the EVT shows $vPath$.

and $nnRad = 0.15 \cdot robot.size$. We set the retracting parameter α to 0.8. For RRT and RRT* we set the growth step Δq to 100% of the scene size. This delivered faster results and only slightly longer paths than a smaller value for Δq.

4.6.2 Evaluation

Sampling strategy: For a comparison of the different sampling strategies we show by way of example the points that are sampled in one single run of the algorithms for the trunk ground plate, test data b). In Figure 38, it can be seen that the algorithms find several possible disassembly paths to different goal positions. The shortest disassembly path has a narrow passage in the middle and is found reliably only by the EVT and EST. The RRT/RRT* covers the entire configuration space with a uniform sampling and so as in this run - rarely finds the shortest path through the narrow passage. It can be seen that the EST found the narrow passage in this run, also tends to cover the large free space inside the car or to find a disassembly path through a door. This behavior can be seen to a greater extent with the EST-connect, which in principle has the same sampling strategy as the EST and did not find the correct path in this run. Since our EVT always performs dense sampling only at the narrow passages, it fulfills

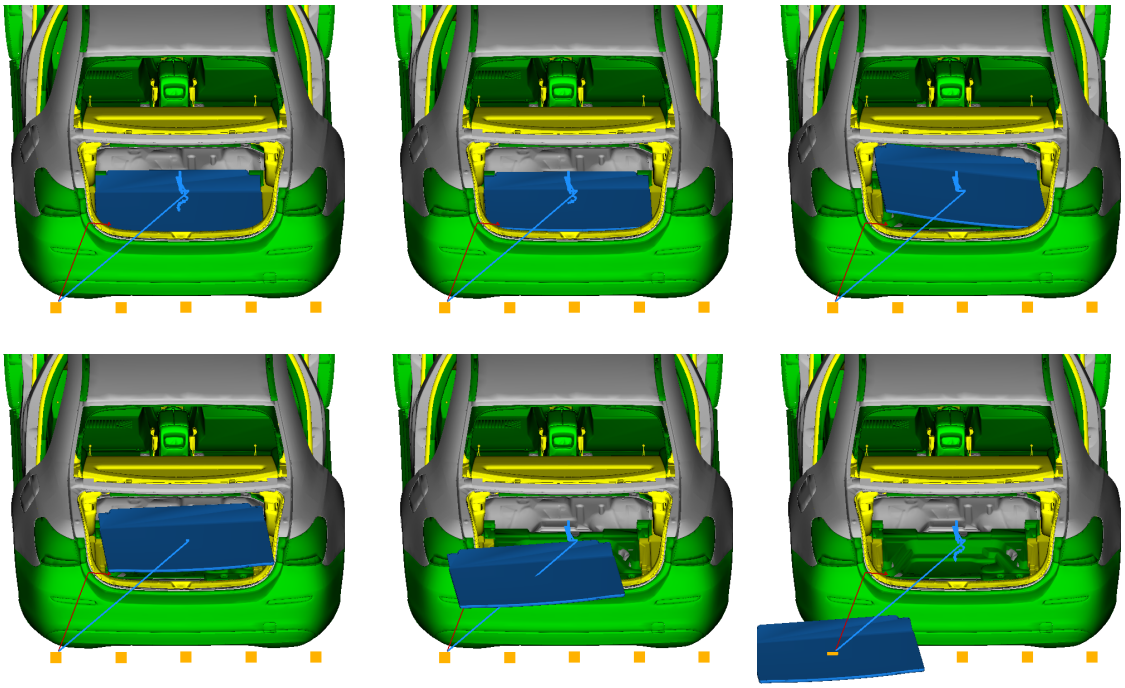

Fig. 39: The found disassembly path is shown in discrete steps for the blue-highlighted part. Reading order is from top left (initial position) to bottom right (goal position).

requirement (*i*). A disassembly path found for the part from Figure 37 b) is shown in Figure 39.

Path length: Next we measure the path lengths obtained by the different algorithms, see Figure 40-top for a graphical evaluation via a box plot. Our EVT algorithm delivered for all parts - except example b) - the shortest median length. Even in example b) the EVT had the shortest 75 quantile length. The low variance among the different path lengths from the EVT for all parts is testament to its stability. In our experiments, the RRT*'s path lengths were only negligibly shorter than the path lengths from the RRT. Since the EVT always delivered short paths and it is based on the GVD, it fulfills the requirement (*ii*) regarding path length and clearance. The superiority of our EVT algorithm compared with the state-of-the-art algorithms can be explained as follows: If a part is assembled 'inside' of the car, there are many different paths it can take to the 'outside', e.g. through the windshield, doors or trunk. Due to random sampling each motion planner has a probability to find a disassembly path through one of the aforementioned ways. The different ways through the different exits differ in terms of length, so the motion planner sometimes - but not always - finds the shortest disassembly path. All the motion planners used have this problem in common.

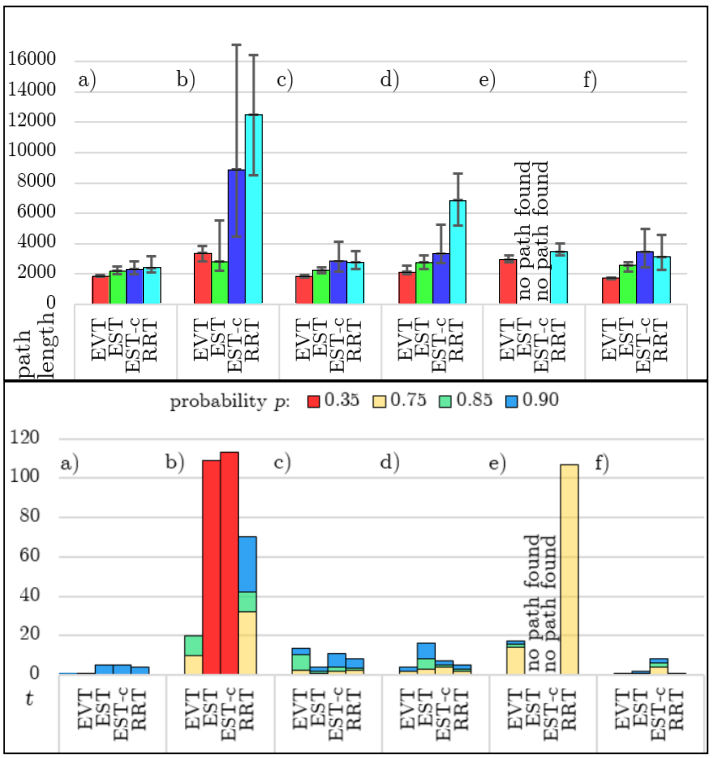

Fig. 40: Top: Computed path lengths for EVT, EST, EST-connect, RRT. The bar height is the median value, the spikes below/above represent the 25/75 quantile. Bottom: The time $t(p)$ to ensure a given pathfinding probability $p = (0.35, 0.75, 0.85, 0.9)$.

They differ only in terms of which way will likely be found. The RRT will likely find an easy to find disasembly path; the EST will find a diassembly path with narrow passages, and so on. Our EVT searches only near the VP which is at least a very good estimation for the shortest disassembly path. Consequently, the EVT can only find a very short disassembly path. This results in short disassembly paths with a very small variance in the path length.

Reliable running time: In our last experiment (see Figure 40-bottom), we consider different probabilities p. For every algorithm we measure for each part the time bound t for which $p\%$ of the runs found a feasible path. The measured overall reliable running time $t(p)$ of the algorithm is therefore the maximum across all these time bounds. We explain this in more detail using the example of the RRT, whereby Figure 41 shows how

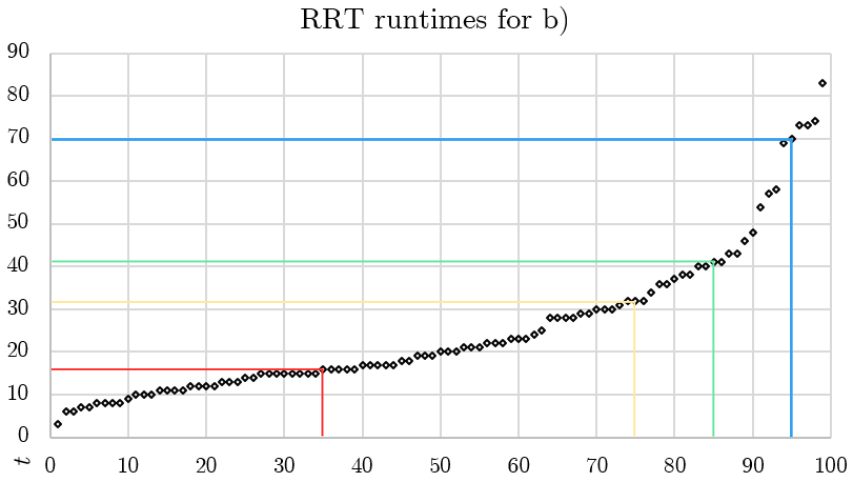

Fig. 41: The ordered run times that the RRT needed for the example in b). The reliable runtimes for $p = (0.35, 0.75, 0.85, 0.9)$ are represented with the colors from Figure 40.

we calculated the reliable running time $t(p)$. For each run (x-axis), the diagram shows the time (y-axis) when the RRT found a feasible disassembly path. The RRT found a feasible disassembly path in 99 of the 100 runs. The reliable running time $t(0.85)$ for part b) is 42 seconds because, in 85 out of 100 runs, the RRT needed at most 42 seconds to find a feasible disassembly path. The overall reliable running time for a possibility p is now the maximum time needed across all examples a) - f). The overall reliable running time is important in the context of ASP because a framework must set a maximum time limit for a motion planner until it terminates - even without finding a feasible disassembly path. Consequently, an ASP framework needs the maximum time limit for each assembled part that cannot yet be disassembled. Therefore, a motion planner is needed that, for each assembled part, finds a feasible disassembly path with a high probability and in a short time.

Our EVT has a reliable running time of $t(0.85) = 20$ seconds because it delivered a path for each part in at least 85 out of the 100 runs within 20 seconds. This demonstrates the robustness of the EVT for finding a disassembly path for different robot-obstacle scenarios reliably and in a short period of time. Neither the EST nor the EST-connect is reliable at all because, for part e), neither were capable of finding a path. The very poor performance of these two motion planners can be explained by the many narrow passages in many different directions. However, only one direction includes a feasible disassembly path. Thus, the EST/EST-connect wastes most of its time exploring narrow passages that do not offer any disassembly paths. Even disregarding this fact, EST/EST-connect already have, for a low probability value, a high reliable running

Table 7: A detailed comparison of the FAR planning phase from Table 10 for the two motion planners EVT and RRT.

tier	suc. queries		time [s]		∅ path length	
	EVT	*RRT*	*EVT*	*RRT*	*EVT*	*RRT*
0	357	354	736	1,320	1,876	2,106
1	69	64	123	281	1,990	2,627
2	47	45	117	200	1,663	2,120
3	55	55	110	72	1,870	1,832
4	25	26	45	75	1,776	2,350
5	26	26	87	164	2,011	1,856
6	26	26	69	32	2,275	2,352
7	8	10	52	41	2,006	2,243
$\sum$	617	610	1,420	2,185	1,892	2,163

time $t(0.35)$ of more than 100 seconds. The RRT achieved a reliable running time of $t(0.75) = 105$ seconds. The RRT* was run for 120 seconds for all parts and, within this time bound, could ensure only a pathfinding probability of 0.35. This is because the RRT* generally behaves like the RRT. However, it takes a long time to rewire the tree. The RRT* thus generally needs more time to find a feasible disassembly path than the RRT. All in all the EVT had by far the best reliable running time for the high pathfinding probability $p = 0.85$, fulfilling requirement (iii).

All parts: In addition, we compared the EVT with the RRT for all assembled parts: this is shown in Table 7. We discuss this table in detail later in the evaluation of our ASP framework (Section 5.4). However, the basic conclusion of the experiment with all assembled parts is clear: Our EVT algorithm finds a feasible disassembly path for more parts, is faster and delivers shorter disassembly paths than the RRT. We compared the EVT only with the RRT because, as already stated, the RRT was the best state-of-the-art motion planner.

5

ASSEMBLY SEQUENCE PLANNING

This chapter, focuses on our main topic: assembly sequence planning (ASP) for real-world CAD data. We will first formally introduce ASP and discuss the various special requirements in relation to complex real-world CAD data. We discuss the representation of the disassembly sequences found for the assembled parts. Common representations like AND/OR-graphs need exponential storage and so are not practical for big data sets. We solve this problem with our novel *assembly priority graph (APG)*. The APG is a compact, meaningful, easy-to-read and extendable blocking graph capable of representing numerous disassembly sequences. We will then present the state-of-the-art frameworks and discuss their shortcomings for real-world data. Finally, we will introduce our ASP framework, the first one capable of handling real-world data in the context of ASP. Our framework builds on the two previous chapters. We will therefore describe in detail how we use the GVD (Chapter 3) and EVT motion planner (Chapter 4) for our framework. The outstanding properties of our ASP framework are:

(*i*) handling of flexible (sub)parts in the NEAR planning phase,

(*ii*) finding of short paths in the FAR planning phase,

(*iii*) reducing the number of impossible disassembly queries (IDQ) which gives a huge speed up

(*iv*) and the meaningful and easy to read representation with our APG.

Our experiments show that our framework is capable of handling the test data from Chapter 2. In addition, the experiments show that common ASP frameworks are not capable of handling our real-world data set.

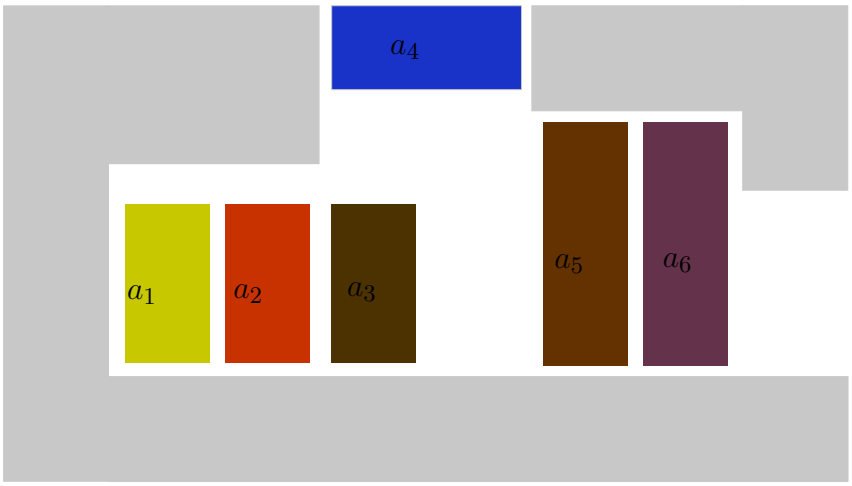

Fig. 42: A two-dimensional example of an assembly. The grey-highlighted parts indicate the body shell; the six colored parts $a_1, ..., a_6$ indicate the assembled parts.

5.1 Assembly Sequence Planning Problem

The aim of ASP is to find at least one feasible assembly sequence if one exists. A feasible assembly sequence is a list indicating a sequence in which all components can be assembled collision-free via a specific assembly path. In our case, we subdivide the given assembly into parts to be assembled and parts to remain in the assembly, the *body-in-white*. A common way of finding feasible assembly sequences is the *assembly-by-disassembly* approach (Mello and Sanderson, 1988) whereby the assembly is disassembled and the resulting disassembly sequence is reverted. Thus, the terms 'assembly' and 'disassembly' are used equivalently, i.e. we search for disassembly sequences and disassembly paths. The advantages of the assembly-by-disassembly approach are described in the following.

The two-dimensional example in Figure 42 shows an assembly that consists of the body-in-white and six assembled parts. One possible feasible disassembly sequence is $a_4 \rightarrow a_5 \rightarrow a_6 \rightarrow a_3 \rightarrow a_2 \rightarrow a_1$ and the corresponding feasible assembly sequence is $a_1 \rightarrow a_2 \rightarrow a_3 \rightarrow a_6 \rightarrow a_5 \rightarrow a_4$. This shows the advantage of disassembly because each assembled part that is successfully disassembled cannot lead to 'dead-ends´ later on. i.e. each part which gets disassembled brings the disassembly process forward. If one starts only with the body shell and assembles one part each after another an assembled part might later block the path for parts that have not yet been assembled. Consider the following (sub) assembly order to be assembled in the body-in-white: $a_2 \rightarrow a_6 \rightarrow a_4$. The parts can be assembled in this order without collision up to this

point, but, it now blocks the other parts and prevents them from being assembled. This results in a time-consuming brute-force algorithm. For the rest of this work, therefore, we use the disassembly process and simply reverse the disassembly sequence to obtain the assembly sequence.

In addition, we assume the following properties: The assembly can be disassembled *monotonous*, i.e. it is possible to disassemble one part after another without moving another part in between or to disassemble a set of parts as a subassembly. A car contradicts this assumption; the doors, the bonnet and the boot lid need to be opened and closed several times during the assembly process. We therefore removed this parts in our data set. Subassemblies play a significant role in the industrial assembly process because they allow parallel assembly steps. However, we have not seen any cases in our data set where subassembly detection is needed. We refer to section 6.2 where we discuss some ideas for non-monotonous disassembling.

The aforementioned disassembly sequence is just one of many. An *assembly precedence graph* represents all feasible disassembly sequences. Michniewicz (2019) shows a number of representations for the assembly precedence graph such as the AND/OR graph and diamond graph. With respect to the number of assembled parts, the assembly precedence graph generally consists of an exponentially high number of many disassembly sequences (Jiménez, 2013). The respective disassembly precedence graph for the example in Figure 42 is shown in Figure 43. Every path from the root a_4 to a leaf is a feasible disassembly sequence. The disassembly precedence graph shown therefore consists of ten different disassembly sequences. Even for mid-sized data sets, the calculation time for an assembly precedence graph can increase by several orders of magnitude. Therefore, the minimum requirement for solving the ASP task is to find at least one feasible disassembly sequence. The more disassembly sequences are found, the more meaningful the solution.

We calculated the aforementioned assembly sequence by looking at the still assembled parts and then removing them one by one whenever the respective part can be removed without collisions. So, we calculated a disassembly path for each assembled part (see Definition 9). To ensure that an assembled part can be disassembled in any given state, an ASP framework needs to calculate a disassembly path at some point for each assembled part. Next, we define the ASP problem.

Definition 13 (*Disassembly Sequence I*)

Given an assembly $\bar{A} := (O, A)$ *which consists of the body shell* O *and all assembled parts* A *and a set of goal points* B*. Then a* disassembly sequence *seq for the assembled*

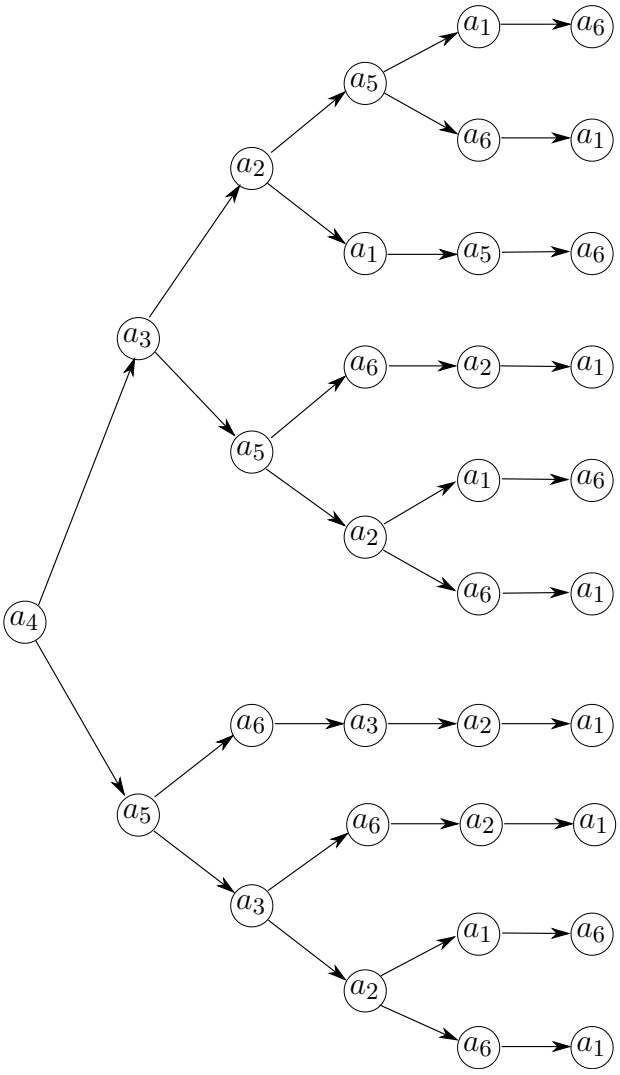

Fig. 43: The disassembly precedence graph for the example from Figure 42.

parts A is an ordered list $seq(A) := \{a'_1, ..., a'_n | a'_i \in A \; \forall i \in \{1, ..., n\}\}$ of all assembled parts A such that there exists for each part $a_i \in seq(A), i \in \{1, ..., n\}$ a disassembly path for the following motion planning problem (see Definition 10): the dynamic object is a_i and the set of obstacles is $\bar{O} := \{O\} \cup \{a_j \in seq(A) | j \in \{i+1, ..., n\}\}$ and the set of goal points is B.

Definition 14 (*Assembly Sequence Planning Problem I*)

Given an assembly $\bar{A}$. Then the ASP problem is to find at least one feasible disassembly sequence for the assembled parts A.

This formal definition works for academic examples. However, as mentioned in Section 4.2, the definition of a feasible disassembly path in the close environment of an assembled part needs to be broadened. Therefore, we broadened the definition of motion planning (Definition 11 and Definition 12) such that they are suitable for real-world data. Consequently, we can now broaden the ASP definitions such that they are also suitable for real-world data sets, i.e. we use the broadened motion planning definitions for our ASP definitions as well. Definition 15 differs only from Definition 13 by the used motion planning problem definition, i.e. we use now the motion planning Definition 12 which allows some controlled collision in the NEAR planning phase.

Definition 15 (*Disassembly Sequence II*)

Given an assembly $\bar{A}$ and a set of goal points B. Then a disassembly sequence *seq for the assembled parts A is an ordered list $seq(A) := \{\bar{a}_1, ..., \bar{a}_n | \bar{a}_i \in A \; \forall i \in \{1, ..., n\}\}$ of all assembled parts A such that there exists for each part $a_i \in seq(A), i \in \{1, ..., n\}$ a disassembly path for the following motion planning problem (see Definition 12): the dynamic object is a_i and the set of obstacles is $\bar{O} := \{O\} \cup \{a_j \in seq(A) | j \in \{i+1, ..., n\}\}$ and the set of goal points is B.*

Definition 16 (*Assembly Sequence Planning Problem II*)

Given an assembly $\bar{A}$. Then the ASP problem is to find at least one feasible disassembly sequence after Definition 15 for the assembled parts A.

5.1.1 Assembly Priority Graph

One reason that the assembly precedence graph explodes is that the number of disassembly sequences increases exponentially with the number of parts that can be disassembled independently of each other. Consider here the parts a_1, a_2, a_3 and a_5, a_6 from the example shown in Figure 42. If a_4 is disassembled, the two subgroups a_1, a_2, a_3 and a_5, a_6 can be disassembled independently of each other, i.e. it is irrelevant for a_5 and a_6 whether or not some of the parts a_1, a_2, a_3 are still assembled. However, each permutation of the disassembly order between the subgroups a_1, a_2, a_3 and a_5, a_6 results in a new disassembly sequence which needs its own representation in common graphs like that shown in Figure 42. Thus, the single piece of information that these two subgroups can be disassembled independently of each other implies that many disassembly sequences are possible. Summarized, the main problem with represen-

tations that list all disassembly sequences is that even for one disassembly path per assembled part the number of possible disassembly sequences explodes. Therefore, indicating only the parts that need to be disassembled in order for an assembled part to be disassembled along a specific disassembly path contains all the required information about possible disassembly sequences. The representation of the parts that need to be disassembled first along a disassembly path is very slim. For example, a blocking graph represents - depending on the underlying disassembly path for a_3 - the information that a_3 can not be disassembled if a_4 is still disassembled. This is the general idea of *blocking graphs* (Wilson, 1992). However, the blocking graph in its original definition cannot represent the blocking information for different disassembly paths in one graph. We can solve this problem thanks to our novel *assembly priority graph (APG)*, which represents all disassembly blocking relationships for all parts and all disassembly paths in one compact graph. We also show that the size of the APG grows only quadratically with respect to the number of assembled parts and linearly with respect to the number of disassembly paths that are represented. Our APG is therefore a combined graph containing the same information as the exponentially growing assembly precedence graphs.

For each assembled part and the different disassembly paths, our *assembly priority graph (APG)* indicates all parts that need to be disassembled first. More precisely, our APG is a directed hypergraph $HG := (V, E)$ (Ausiello and Laura, 2017) where each part defines a node. A hyperedge $h_{dp} := (a_{start}, \{a_{end_1}, ..., a_{end_n}\})$ represents for a given disassembly path dp the priority relationship of the components that must already be removed for collision-free disassembly. So, a_{start} can be successfully disassembled along dp if $a_{end_1}, ..., a_{end_n}$ are disassembled. For the disassembly paths which are shown in Figure 44 see Figure 45 for the corresponding APG. The APG represents both disassembly paths for a_1. The dashed disassembly path is represented with the directed dashed hyperedge $(a_1, \{a_2, a_3, a_4\})$ and the continuous disassembly path with the continuous hyperedge $(a_1, \{a_2, a_3, a_5, a_6\})$. The loop-edge at a_4 indicates, that a_4 has no collisions if disassembled along the corresponding disassembly path. Another alternative to the loop-edge is to draw an empty edge, i.e. just draw no edge. However, the loop-edge is needed if two or more different disassembly paths are available: one with no collisions and another one with collision. Then, the empty edge cannot be read out from the APG. We therefore define the hypergraph and our APG as described below.

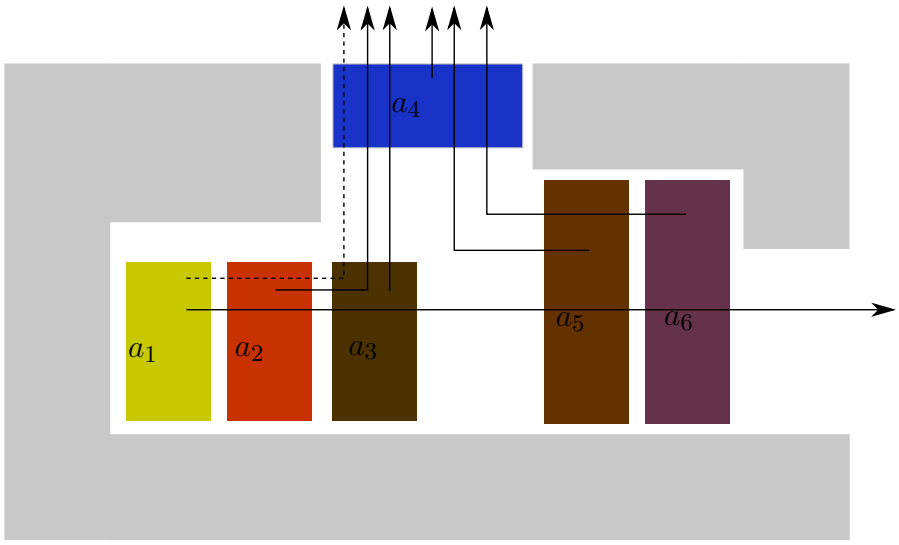

Fig. 44: A two-dimensional example of an assembly. The grey-highlighted parts indicate the body shell, the six colored parts indicate the assembled parts. The (dashed) arrows represent possible disassembly paths.

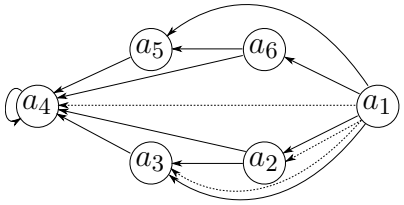

Fig. 45: The disassembly precedence graph for the example from Figure 44.

Definition 17 (*Hypergraph*)

A directed hypergraph $HG := (V, E)$ *is an ordered tuple, where* V *indicates the set of vertices and* E *the set of directed hyperedges. A directed hyperedge* $(a_{start}, a_{End}) =: e \in E$ *with* $a_{start} \in V, a_{End} \subseteq V$ *indicates the directed edges* $(a_{start}, a_{end})\ \forall a_{end} \in a_{End}$*. A directed hyperedge therefore has one starting vertex and up to* $|V|$ *end vertices.*

Definition 18 (*Assembly Priority Graph*)

Given an assembly $\bar{A} := (O, A)$ *which consists of the body shell* O *and all assembled parts* A *and for each assembled part* $a \in A$ *a set of disassembly paths. Then the* $APG := (V, E)$ *is a directed hypergraph where the set of nodes* $V := A$ *is equal to the assembled parts. For each disassembly path, the APG contains one hyperedge* e*. Let* $dp := dp_{near} \cup dp_{fear}$ *be a disassembly path (see Definition 11) for* $a \in A$*. Furthermore, let* $a_{End} \subseteq A \setminus \{a\}$ *be the set of assembled parts with which* a *collides if disassembled along* dp*, except that assembled parts which collide with* a *only within the tolerated*

collision in the NEAR planning phase. If $|a_{End}| = 0$ *we set* $e := (a, \{a\})$ *otherwise* $e := (a, a_{End})$.

Next, we analyze the size of our APG.

Theorem 3

Let $\bar{A}$ *be an assembly and* $APG := (V, E)$ *a corresponding assembly priority graph. Assume that each assembled part* $a \in A$ *has at most* n *different disassembly paths. Then the size of the assembly priority graph* $size(APG) := |V| + \sum_{e \in E} |e|$ *with* $|e| := 1 + |a_{End}|$ *is at most* $|A| + n|A|^2$.

Proof: Each assembled part a is represented with one vertex. So it is $|V| = |A|$.

A hyperedge $e := (a_{start}, a_{End})$ indicates with the set a_{End} which parts have to be removed for a collision-free removal along the associated disassembly path. It is not possible for a part to have to remove itself first. Thus, we conclude $|a_{End}| \leq |A| - 1$ and therefore $|e| \leq 1 + (|A| - 1) = |A|$. By assumption each assembled part a has at most n disassembly paths and therefore each vertex has at most n outgoing directed hyperedges. Thus, $|E| \leq n|V| = n|A|$. So it is $\sum_{e \in E} |e| \leq |E| \cdot |A| \leq n|A| \cdot |A| = n|A|^2$ which leads to $size(APG) = |V| + \sum_{e \in E} |e| \leq |A| + n|A|^2$. ■

Theorem 3 shows that the size of the APG is with respect to the number of disassembly paths found for each part quadratically bounded. The size of the representations that list all disassembly sequences, e.g. an AND/OR-graph, is generally exponential. This result shows that our APG is superior compared to common assembly precedence graphs.

5.2 ASP Frameworks

5.2.1 Overview

An ASP framework receives an assembly as its input and delivers at least one feasible disassembly sequence as its output. A conventional solution (Ebinger *et al.*, 2018), (Yu and Wang, 2013), (Kardos and Váncza, 2017), (Zhang *et al.*, 2017) is the following: A loop always iterates across all assembled parts and attempts to find a feasible

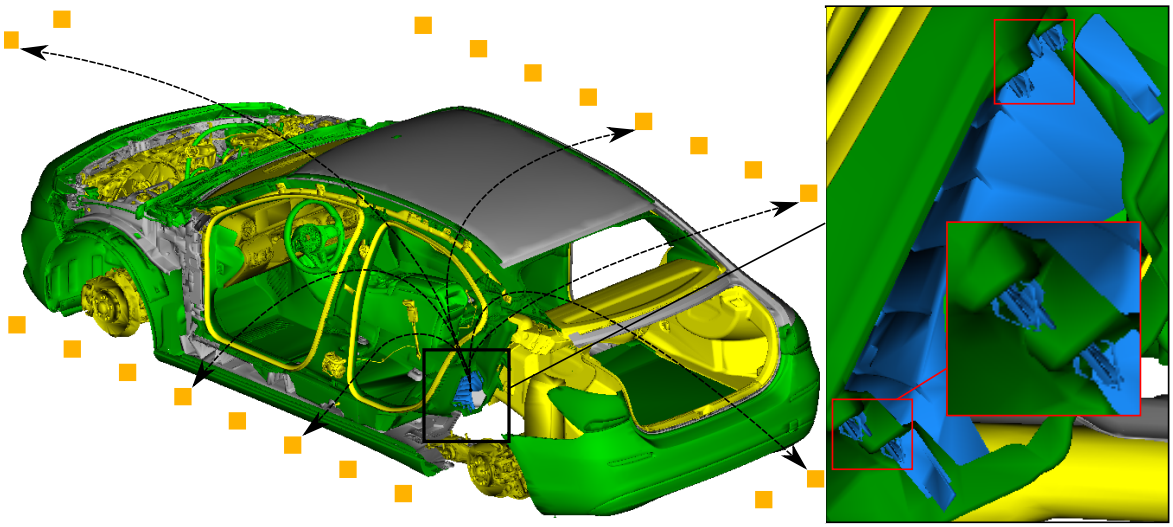

Fig. 46: The data represents a subset of the Mercedes-Benz A-Class. The light-grey parts belong to the body shell, while the other colors represent the assembled parts. The surrounding orange squares represent goal positions. The dashed arrows show possible feasible disassembly paths for the blue-highlighted covering part. This covering part is fastened with the red-highlighted flexible clips shown on the right.

disassembly path for each assembled part until all parts are disassembled. In general, depending on the number of assembled parts, the loop executes a quadratic number of disassembly queries. Each part can be disassembled only once, however, which means that the majority of motion planning attempts are unsuccessful because a part to be checked is blocked by other still assembled parts and so cannot yet be removed. We call these *impossible disassembly queries (IDQ)*. The quadratic runtime of this brute-force loop is too slow for a large number of assembled parts. So for an approach that is viable in practice, the number of IDQ needs to be significantly reduced. We present two algorithms that quickly filter out many of those IDQ. Reducing the IDQ is beneficial for another reason. Motion planning queries for parts that cannot yet be disassembled are more time-consuming than motion planning queries with a successful outcome. This is because if no disassembly path exists, the motion planner used terminates only after exceeding the given maximum time limit. Each IDQ therefore requires the maximum time available. Motion planning queries with a positive outcome require by definition less time.

As already mentioned, the actual computation of a disassembly path for a part is a (rigid body) motion planning task. Here, most ASP works address only linear paths (Ou and Xui, 2013), (Su, 2006), (Yu and Wang, 2013). Since this is very often not sufficient in real-world scenarios, some works (Ebinger *et al.*, 2018), (Aguinaga *et al.*, 2008) use the sampling-based motion planner 'Rapidly-exploring random tree' (RRT) (Lavalle,

1998) for finding feasible disassembly paths. All previous works for ASP are tested on small and, in most cases, only in an academic context. When it comes to complex real-world scenarios, two main difficulties occur. First, as shown in the example in Figure 46, it is often impossible to find feasible disassembly paths without addressing flexible (sub)parts like fastening elements. Fastening elements (e.g. clips) almost always cause collisions with the surrounding geometry *near* the installed position. In reality, they can be 'unfastened' only if they are deformed. Second, *far* from the installed position, the large workspace in complex real-world scenarios permits many different feasible disassembly paths for each assembled part. These disassembly paths differ in terms of length and clearance. Therefore, an ASP framework should be capable of determining optimized disassembly paths. To cover both these requirements, we subdivide the motion planning process into two disjoint phases: the NEAR- and FAR planning phase (see also Section 4.2 and Section 4.3).

We summarize the requirements for an ASP framework for complex real-world scenarios as follows:

(i) treat flexible (sub)parts like clips in the NEAR planning phase,

(ii) quickly find optimized paths in the FAR planning phase,

(iii) keep the number of IDQ low,

(iv) extract a meaningful representation of the disassembly sequences.

5.2.2 Related Work

One of the first works (Mello and Sanderson, 1988) covering ASP introduced the theoretical basis. Among the main tasks are the definition of ASP and the representation of assembly precedence graphs. A number of works cover the combinatorial and geometric perspective (Jiménez, 2013), (Kavraki *et al.*, 1993), (Ghandi and Masehian, 2015). They show that ASP involves many NP-hard tasks such as motion planning and subassembly detection. It is also shown, that the combinatorial explosion of the assembly precedence graph is not manageable in practice even for mid-sized examples. Next, we provide an overview of the current state-of-the-art regarding requirements (i) - (iv).

Most ASP frameworks consider only linear disassembly paths (Ou and Xui, 2013), (Su, 2006), (Yu and Wang, 2013), (Kardos and Váncza, 2017), (Zhang *et al.*, 2017), (Hadj *et al.*, 2018), (Pintzos *et al.*, 2016). However, some works (Ebinger *et al.*, 2018), (Aguinaga *et al.*, 2008) use the sampling-based motion planner 'Rapidly-exploring random tree' (RRT) (Lavalle, 1998). An RRT is capable of computing arbitrary motions including rotations. Nevertheless, the RRT cannot handle flexible (sub)parts (*i*) and does not deliver optimized paths (*ii*). Masehian and Gahndi (2021) tackle the problem of flexible subparts by creating an *Assembly Stress Matrix* which indicates the stress on a part during the disassembly process. However, they need the material properties and consider only the disassembly directions along the x, y, and z-axes. Moreover, they use the FEM software *Abaqus*^{TM} (*Abaqus*^{TM} n.d.) to simulate the deformation, which is extremely time-consuming. In summary, no ASP framework exists that fulfills the motion planning requirements (*i*) and (*ii*).

However, algorithms exist outside the context of ASP that deal with problems (*i*) and (*ii*). See Section 4.1.1 and Section 4.2.1 for a general discussion of related work in the field of motion planning.

Finding an optimized path is already widely discussed in the context of motion planning. The extension RRT* (Karaman and Frazzoli, 2011) of the RRT returns a path that optimizes a user-defined criterion such as path length and so generally fulfills (*ii*). However, the RRT* algorithm is much slower than the RRT. Our 'Expansive Voronoi Tree' (EVT) from Section 4.5 is very fast and also delivers optimized paths. However, the EVT requires a runtime-intensive general Voronoi diagram of the complete scene for these benefits. In the context of ASP, however, the runtime costs for the general Voronoi diagram are distributed across all assembled parts. We can therefore demonstrate, that the EVT motion planner is ideally suited for the FAR planning phase.

Only a few works exist that focus on reducing the number of disassembly queries within an ASP process (*iii*). The authors of Hadj *et al.* (2018) assume that fasteners can always be disassembled and so they search only for disassembly paths for non-fastener parts. However, they need preliminary information about which part is a fastener and the assumption that fasteners can be disassembled in every shoring situation is generally not correct. For example, a screw may be longer than the required clearance allows, i.e. the screw hits the back/top during the screwing process. Another work (Popescu and Iacob, 2013) presents an approach that first searches for a disassembly path for assembled parts that are assembled on the 'outside'. However, it is generally not the case that parts on the outside have to be removed before those on the inside.

The structuring or meaningful representation of the disassembly paths found in an ASP framework (*iv*) is discussed by Pintzos *et al.* (2016). They present the so called *search strategy by tiers*. This strategy starts with the complete assembly and in each step removes all parts that can be disassembled independently of each other for the actual shoring state. This proceeding groups all parts in hierarchical tiers which results in a meaningful and easy-to-interpret disassembly order. Our APG also divides the assembled parts into hierarchical tiers.

Like our APG,Wilson's *blocking graph* (Wilson, 1992) contains a node for each assembled part and indicates with its edges the parts that collide for a given disassembly motion with a part. Each blocking graph represents only one disassembly motion, however, which is why many graphs are needed to represent all the different disassembly paths.

5.3 Our Framework

5.3.1 Overview

We will first provide a rough overview of how our ASP framework works, followed by a detailed explanation. The framework can be described with three algorithms (see also the pseudocode). The main-loop (Algorithm 13) iterates over all still assembled parts and at the start of each iteration, calculates the GVDG. It then, uses the GVDG and our novel collision perceiving algorithm to decide if a part should be checked for disassembly (Algorithm 14). When the check is executed (Algorithm 15), it first searches for the disassembly path in the NEAR planning phase and then, if successful, on the basis of the associated VPs for disassembly paths for FAR planning. All the parts that are disassembled in the same iteration step are categorized in the same *tier*. This means that these parts can be disassembled independently from each tier and the following tiers. A detailed explanation of the specific steps is provided below.

Algorithm 13 APG calculateAPG(A, O)

1: $assParts \leftarrow A$
2: **while** $assParts \neq \emptyset$ **do**
3: $\quad calculateGVDG(assParts, O)$
4: $\quad$ **for all** $part \in assParts$ **do**
5: $\quad\quad$ **if** $disassemblyPossible(part)$ **then**
6: $\quad\quad\quad findDisassemblyPath(part)$
7: $\quad removeAllDisassembledParts(assParts)$
8: **return** $createAPG(A)$

5.3.2 calculateAPG

Algorithm 13 shows the main function *calculateAPG*. It receives all assembled parts A and the body shell O as its input. First, we assign all assembled parts A to the set of still assembled parts $assParts$. The while-loop terminates when all assembled parts are disassembled (line 2). At the beginning of each iteration, we calculate the general Voronoi diagram (GVD) for the body shell and all still assembled parts and extract the general Voronoi diagram graph (GVDG) from this (line 3). We calculate the GVDG with the OTI approach from Section 3.2.3. We search for each still assembled part up to two VPs, one path leading to a front/back goal position and another to a lateral goal position (see Figure 47 for a visualization of the VPs). Setting two different VPs increases the probability of finding a feasible disassembly path for the FAR planning phase and, if both VPs lead to a feasible disassembly path, it increases the variety of the APG.

The for-loop (line 4) checks for all still assembled parts, to see whether a feasible disassembly path can be found (line 5 and Algorithm 14). If a disassembly path exists in principle, we search for a disassembly path (line 6 and Algorithm 15). After the for-loop, all disassembled parts are removed from the still assembled parts $assParts$ (line 7). Our procedure groups all the disassembled parts in this step into one hierarchical tier, following the procedure of Pintzos *et al.* (2016). This results in a simple but meaningful interpretation of assembly precedence relations: All parts assigned to one tier can be removed independently of all parts contained in the same or a subsequent tier. Finally, the APG (see Section 5.1.1) is created and returned.

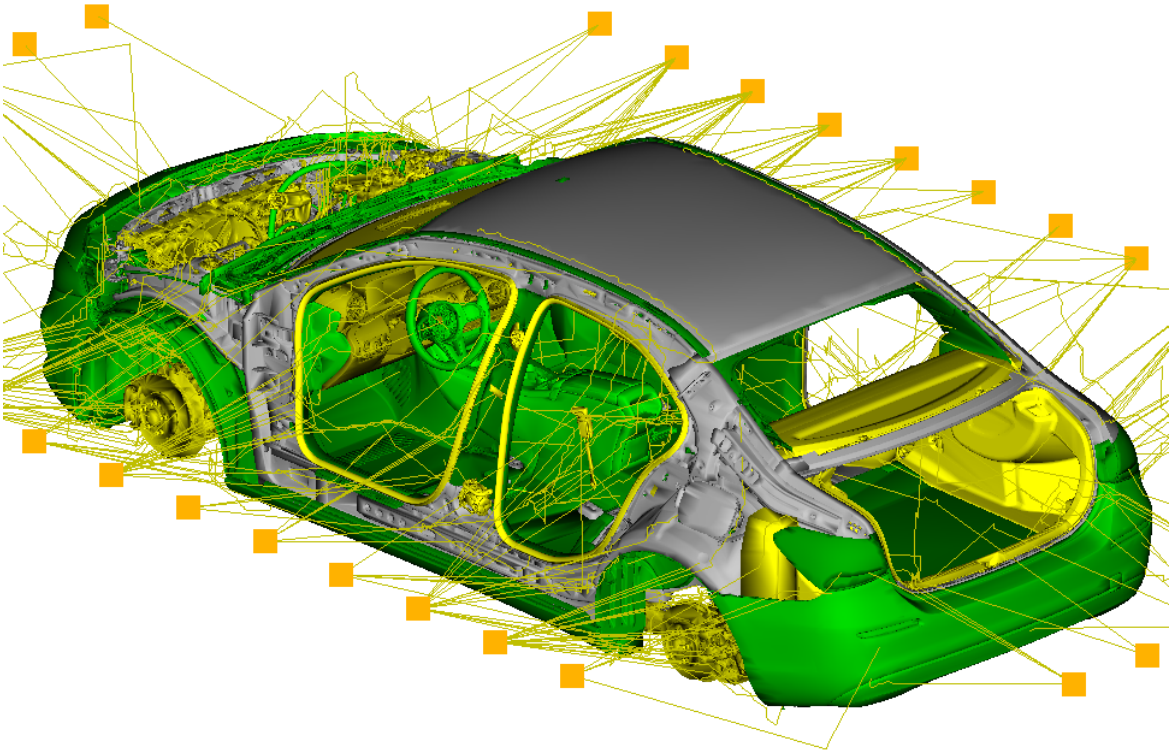

Fig. 47: Our test data with VPs for assembled parts that lead to the orange goal positions B.

We will now describe how we determine a hyperedge $h_{dp} := (a, a_{End})$ for a part a that is disassembled via the disassembly path dp. In short, we push the part a along dp and collect all the parts that collide with a. A detailed explanation is provided below. Given an assembly $\bar{A} := (O, A)$ with its body shell O and the assembled parts A. Let $a \in A$ be a part with a corresponding disassembly path $dp := dp_{near} \cup dp_{far} := \{c_{init}, c_1, ..., c_n, c_{goal}\}$. Assume that a is in tier $i \in \mathbb{N}_0$ and the parts that are in tier $j < i$ are already disassembled. Then the hyperedge h_{dp} is determined as follows. We apply all configurations of dp to a and check which parts of A' collide with a. The set of parts $A' := \{a' \in A | \text{tier of } a' < i\}$ consists of all parts that are still disassembled. It is important to use the set A' instead of all assembled parts A because in the NEAR planning phase, we allowed some controlled collision. This controlled collision should be ignored from the hyperedge, i.e. the parts that cause the controlled collision must not be disassembled first. So, the set of outgoing nodes of h_{dp} is $a_{End} = \{a' \in A' | \exists c \in dp(c(m) \text{ collides with } a')\}$. For $a_{End} = \emptyset$ we set the self-loop $a_{End} = \{a\}$ to indicate that no parts are in collision.

Algorithm 14 bool disassemblyPossibly($assPart$)

```
1: if assPart.voronoiPath == NULL then
2:     return false
3: for all collPart ∈ assPart.collisionParts do
4:     if collPart ∉ assParts then
5:         return true
6: if assPart.collisionParts == ∅ then
7:     return true
8: return false
```

5.3.3 disassemblyPossible

Algorithm 14 shows the *disassemblyPossible* function. Motion planning queries for which no feasible disassembly path exists are extremely time-consuming, which is why, this function performs a check using two new heuristics to determine whether a disassembly path exists for an assembled part. As shown in our experiments in Section 5.4, this significantly speeds up the running time. The time-consuming search for a feasible disassembly path is performed in Algorithm 15.

Voronoi path existence (lines 1, 2)**:** The GVDG does not always provide a VP for an assembled part. We will analyze why this happens and explain the resulting conclusion. We differentiate between three different cases, as illustrated in Figure 48. In each example a), b) and c), the assembled part a_1 is to be disassembled. In the first case a), the GVDG connects the Voronoi cell of a_1 with the surrounding and thus a VP for a_1 can be found. Furthermore, Algorithm 15 then finds a disassembly path for a_1. In b), the GVDG also provides a VP for a_1. However, Algorithm 15 will not find a disassembly path for a_1 because the gap between O and a_2 is too narrow. In the last case c), the assembled part a_1 is completely enclosed by its surrounding parts O and a_2. Therefore, the Voronoi cell of a_1 does not have a connection to the outside and cannot deliver a VP to a goal position. In this case $assPart.VP ==$ NULL and Algorithm 15 is not called at all because if no VP exists, no disassembly path exists. We therefore check to determine whether the part to be disassembled has a VP (line 1) and return false (line 2) if no VP and, in turn, no disassembly path exists (case c)). We use this method to filter out all enclosed assembled parts.

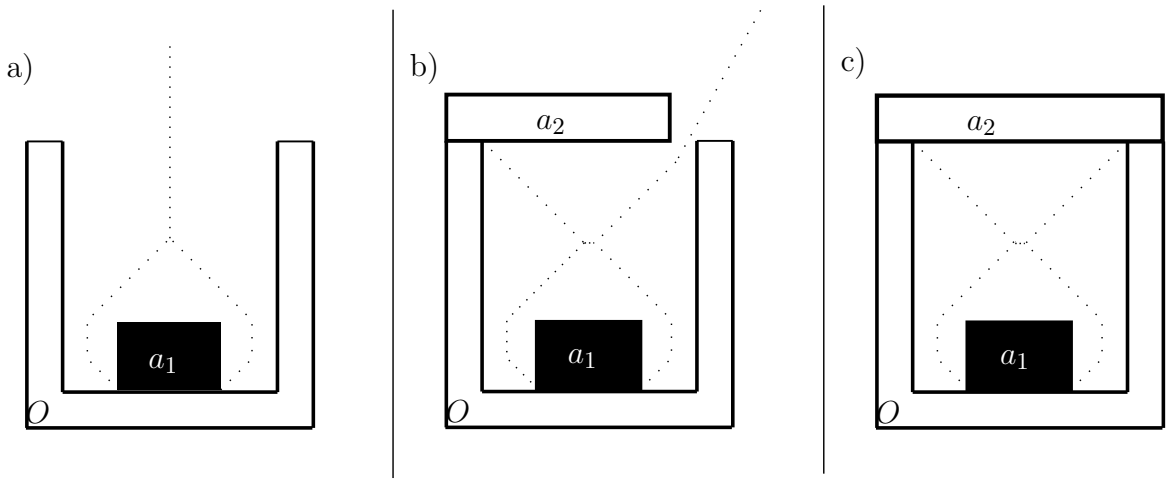

Fig. 48: Three different assembly states are shown in a), b) and c). The assemblies consist of the body shell O and assembled parts a_1, a_2. The corresponding GVDGs are represented with the dotted line.

Collision perceiving (lines 3 - 7)**:** The case shown in Figure 48 b), that a VP but not a disassembly path exists, occurs inevitably in complex scenarios. However, in this case, a motion planner in Algorithm 15 attempts in vain to find a feasible disassembly path. The motion planner starts its sampling procedure and terminates after a given time limit. Requiring the full time available without identifying a feasible disassembly path is a very time-critical aspect for an ASP framework. During this sampling process, the *assPart* collides with its surroundings. We store each assembled part that collides with the *assPart* during this sampling process in the list *assPart.collidingParts*. We now assume the following: If the motion planner has not found a disassembly path after a given time limit, this means that no disassembly path exists in the current assembly state. The part is therefore enclosed by other assembled parts (in b) by a_2) and can be disassembled only when at least one of these colliding parts is disassembled. We therefore check all colliding parts in *assPart.collidingParts* from the previous disassembly attempt (line 3) and attempt to execute Algorithm 15 for *assPart* again only if at least one of these assembled and colliding parts has since been disassembled (line 4, 5). If there are no colliding parts in *assPart.collidingParts* this means that no previous disassembly attempt was made and we also return true (lines 6, 7).

Algorithm 15 void findDisassemblyPath($assPart$)

1: $assParts.collisionParts \leftarrow \emptyset$
2: **if** $!findNEARdisPath(assPart)$ **then**
3: **return**
4: **for** $VP \in assPart.VP$ **do**
5: $findFARdisPath(assPart, VP)$

5.3.4 findDisassemblyPath

Algorithm 15 searches for a disassembly path for each associated VP of the assembled part $assPart$.

First, we clear the collision parts $assParts.collisionParts$ (line 1) from the previous disassembly attempt.

Next, we search for a disassembly path for the NEAR planning phase (line 2). We define the end of the NEAR planning phase as the point at which the assembled part has been moved translationally at least 50 millimeters from its initial position. If no disassembly path was found in the NEAR planning phase, we stop motion planning at this point (line 3). If the part is 50 millimeters away from the initial position, NEAR range planning is considered successful and then we perform FAR range planning, searching for a disassembly path for up to two VPs (lines 4, 5). We will now provide a brief overview of how these motion planners work. See Chapter 4 for a detailed discussion of motion planning.

NEAR motion planning: For the NEAR planning phase, we use the *Iterative Mesh Modification Planner (IMMP)* (Hegewald *et al.*, 2022). This is an alternating method whereby controlled mesh deformations and planning attempts are performed until a maximum number of iterations is reached or a path is found. With the mesh deformations, the method is capable of eliminating any unavoidable collisions due to flexible fastening elements and overpressure on components. For the deformation of fastening elements, the method involves the computation of a saliency map (Hegewald *et al.*, 2021) of the robot that extracts the fastening elements as salient regions. By eliminating any unavoidable collisions, the method is capable of applying the conventional *Expansive Spaces Tree (EST)* (Hsu *et al.*, 1997) motion planner to compute a collision-free disassembly path.

Table 8: Three frameworks with different motion planners for the NEAR- and FAR planning phase.

Framework	NEAR	FAR
①	IMMP	EVT
②	IMMP	RRT
③	RRT	RRT

FAR motion planning: For the entire FAR planning phase, we use our *Expansive Voronoi Tree (EVT)* motion planner, which is described in Section 4.5. The main idea behind the EVT is that it samples along the given VP. The basis of the sampling strategy is the EST motion planner which is suitable for narrow passages. The EVT changes the translational sampling of the conventional EST such that it adapts to the clearance situation and goes in the forward direction of the VP. In addition, the VP acts as a magnet that attracts the configurations back to itself. The rotation is randomly sampled. This motion planner finds shorter paths and is faster than common motion planners such as the RRT and EST as shown in Section 4.6.2.

The runtime bottleneck of the EVT, the calculation of the GVD, is distributed across all assembled parts. In the context of ASP, therefore, the EVT is superior to common motion planners such as the RRT in terms of runtime, path quality and path length.

We expanded both motion planners so that they store the colliding parts in $assParts.collisionParts$ for the *collision perceiving* check.

5.4 Experiments

In this section, we evaluate the ASP framework that we presented and compare it with the common ASP frameworks (Ebinger *et al.*, 2018), (Aguinaga *et al.*, 2008) which use an RRT. We compare three different frameworks ①, ② and ③ against each other. See Table 8 for an overview. The three frameworks differ only in the used motion planner for the NEAR- and the FAR planning phase. Below, we will name the specific framework by its number, e.g. ①. These experiments focuses on demonstrating that our ASP framework ① can be used for complex 3D real-world CAD data, which in our case is a representative data set from the automotive industry (see Figure 46). The data set is composed as follows: The body shell consists of 100 parts, on which 661 parts are assembled. Our framework starts with the complete data set and calculates iteratively the individual tiers.

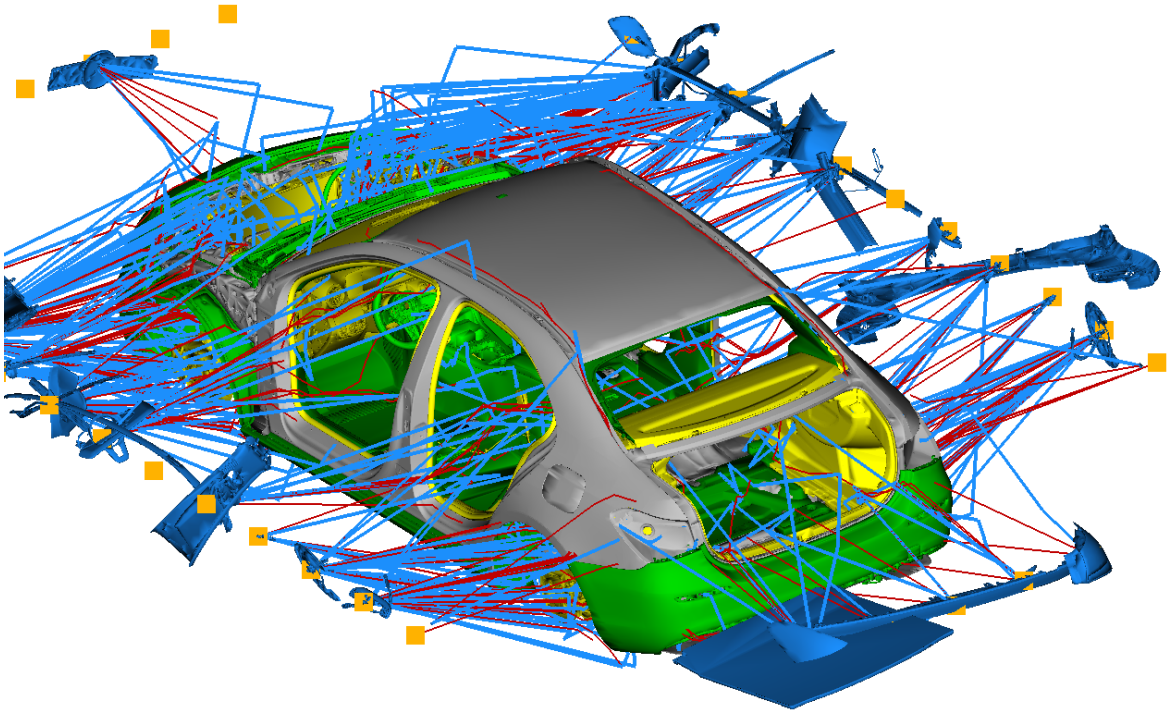

Fig. 49: Our test data surrounded by the blue-highlighted parts from the first tier. The red lines represent the VPs and the blue lines the translational part of the final disassembly paths.

Due to the evaluation from section 4.6, we give the EVT and RRT 20 seconds to find a feasible disassembly path. We allowed the NEAR motion planner IMMP 5 cycles of shrinking and again the subsequent motion planning process to search for a feasible path terminates after 20 seconds. However, the aforementioned shrinking process has no time limit, which means that the IMMP motion planner as a whole has no time limit.

Table 9 and Table 10 show the experimental results of our framework ①. First, we will provide a brief overview of all the results, which are discussed in detail afterward. The 661 assembled parts (column b) were disassembled after the calculation of 8 tiers (column a), which took 105,653 seconds ≈ 30 hours (Table 10 d). See also Figure 49 for a visualization of the first tier $tier_0$ and Figure 50 for all 8 tiers. We will then discuss the results in terms of *calculation time*, *impossible disassembly queries (IDQ)*, *manually disassembled parts*, *motion planning* and *assembly priority graph (APG)*.

Calculation time ① (Table 10): Most of the time (94%) was spent on the NEAR planning phase (Table 10 f and g), of which the majority (84%) was spent on unsuccessful disassembly queries (Table 10 g). With just 3.5% of the overall calculation time, the

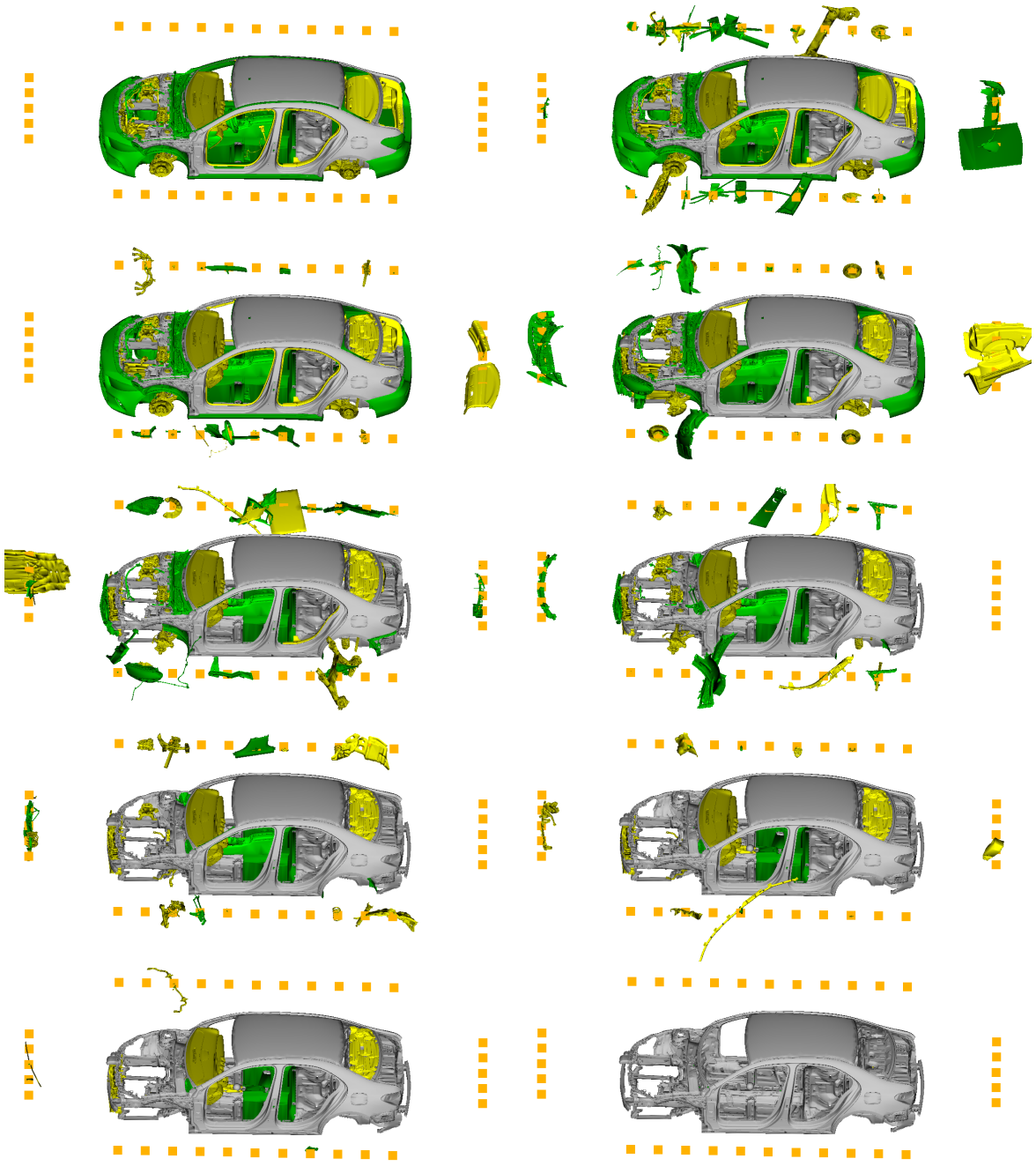

Fig. 50: All found tiers are shown. Starting with the complete assembly (top left), right up to the naked body shell (bottom right).

Table 9: Experiment results of our framework ① for the data set shown in Figure 48.

a	b	c	d	e	f	g
tier	#still ass. parts	#detected imp. dis. queries		#parts in tier	#manually disass. parts	#dis. parts only RRT ③
		VP existence	*coll. perceiving*			
0	661	120	0	357	0	13
1	304	79	84	78	9	10
2	226	58	99	52	5	6
3	174	36	73	56	1	7
4	118	26	48	39	14	0
5	79	15	21	41	15	4
6	38	2	4	26	0	10
7	12	0	0	12	5	0
$\sum$	-	340	334	661	48	50

Table 10: Experiment results of our framework ① for the data set shown in Figure 48.

a	b	c	d	e	f	g	h
tier	#still ass. parts	#parts in tier	tier calc. time [s]	GVDG time [s]	NEAR time [s]		FAR time [s]
					suc	*fail*	
0	661	357	50,995	58	5,433	42,159	2,020
1	304	78	33,359	62	1,438	31,139	435
2	226	52	7,033	59	1,068	5,520	483
3	174	56	5,047	67	818	3,895	191
4	118	39	3,578	76	765	2545	114
5	79	41	2,826	78	745	1740	164
6	38	26	2,257	91	404	1622	105
7	12	12	558	83	38	254	139
$\sum$	-	661	105,653	574	10,709	88,874	3,651

FAR range planning (Table 10 h) had minimal impact. The time taken to calculate the GVDG is insignificant compared with the overall calculation time (Table 10 e).

IDQ ① (Table 9 c and d): The evaluation of the calculation time demonstrates why it is important to reduce the IDQ. The VP existence and the collision perceiving algorithms together detected 674 (340 + 334) IDQ. One unsuccessful disassembly query needed an average of $\approx$ 340 seconds. Therefore, both algorithms combined shortened the calculation time by approximately 229,160 seconds $\approx$ 63 hours which is 67% of the overall calculation time.

Manually disassembled parts ① (Table 9 k): In general, the IMMP NEAR planner is capable of disassembling flexible (sub)parts. However, it has problems with 'completely'

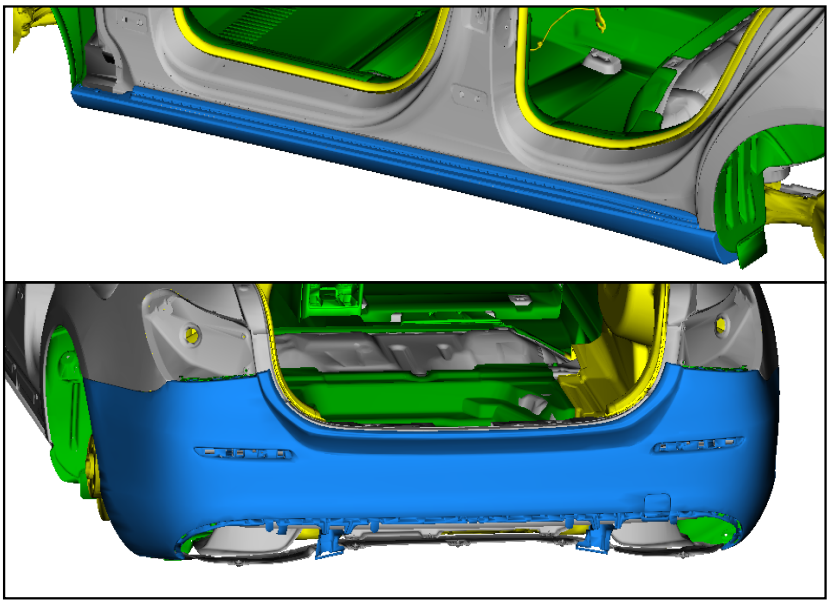

Fig. 51: Two assembled parts where the IMMP motion planner could not find a feasible disassembly path for the NEAR planning phase.

flexible parts such as cables and fabric covers (see Figure 52). Another reason why the IMMP motion planner cannot find a feasible disassembly path for a rigid part is that another part needs to be deformed, i.e. a cover sheet needs to be pushed away slightly to allow a cover plate can pass. This case is shown in Figure 51 top. The two wheelhouses, which laterally enclose the blue-highlighted part need to be deformed slightly. In addition, it is a randomized motion planner that offers no guarantee of quickly finding feasible disassembly paths for parts that have to pass through very narrow passages. This is the case for the blue-highlighted part in Figure 51 (bottom). After each tier calculation, therefore, we added the corresponding parts manually to the current tier and marked them as 'disassembled'. As shown in '#manually disass. parts', our framework could not find a feasible disassembly path for 48 (7%) assembled parts even if one exists.

Motion planning ③ (Table 9 g): Table 9 g '#disass. parts only RRT' shows how effectively common ASP frameworks (Ebinger *et al.*, 2018), (Aguinaga *et al.*, 2008) that use an RRT for the NEAR- and FAR planning phases perform on our data set. The column shows for how many parts in the respective tier an RRT found a disassembly path from its installed position to one of the goal positions. In summary, an RRT ③ found a feasible disassembly path for only 50 of the 661 assembled parts (7.5%). The reason for this poor result is that the RRT does not specifically address flexible

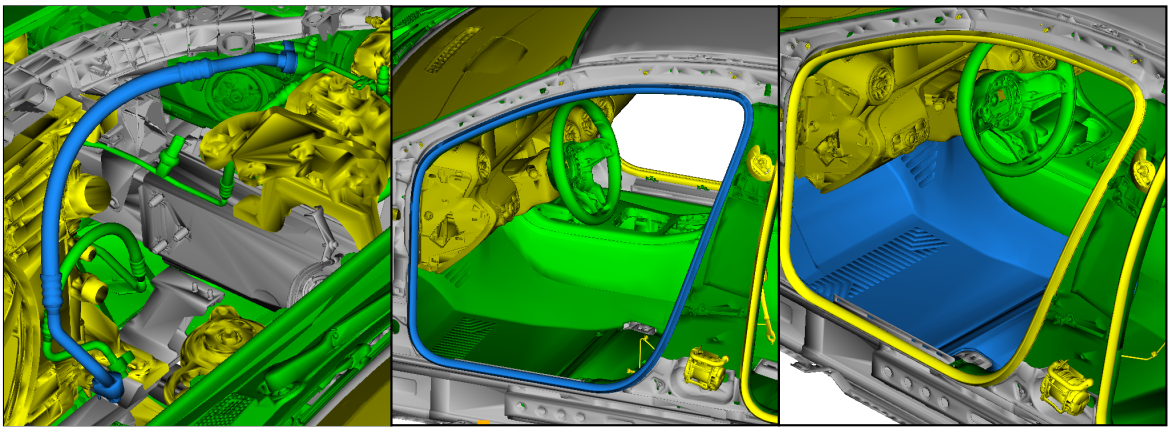

Fig. 52: The three blue-highlighted parts are 'completely' deformable: hose, seal (foam), footmat (fabric).

Table 11: A detailed comparison of the FAR planning phase from Table 10 for the two frameworks ① (EVT) and ② (RRT).

tier	suc. queries		time [s]		∅ path length	
	EVT	*RRT*	*EVT*	*RRT*	*EVT*	*RRT*
0	357	354	736	1,320	1,876	2,106
1	69	64	123	281	1,990	2,627
2	47	45	117	200	1,663	2,120
3	55	55	110	72	1,870	1,832
4	25	26	45	75	1,776	2,350
5	26	26	87	164	2,011	1,856
6	26	26	69	32	2,275	2,352
7	8	10	52	41	2,006	2,243
∑	617	610	1,420	2,185	1,892	2,163

fastening elements. This highlights the importance of a specialized motion planner for the NEAR planning phase.

We compare the two frameworks ① and ②, i.e. the chosen EVT motion planner with an RRT in the FAR range planning phase. In 'suc. queries' Table 11 shows that the EVT found 7 more disassembly paths than the RRT and needed significantly less calculation time for this. Furthermore, the paths which are found by the EVT are 12.5% shorter than the paths from the RRT. Note: The calculation times from Table 10 h 'FAR time [s]' show the time needed for up to two disassembly paths, while the times in Table 11 'time [s]' show only the calculation time needed until one disassembly path was found.

APG: Our extracted APG consists of 661 nodes, with 784 hyperedges each representing a disassembly path. The APG represents with its small size of $size(APG) = |V| + |E| = 661 + 1542 = 2203$ at least 10^{1021} different assembly sequences. This is a lower bound for the number of different assembly sequences which we obtain by assuming that each tier must be assembled one after another and the order of the parts within each tier is arbitrary. This means, that there are $|tier_i|!$ many sub-assembly sequences for assembling each tier, with $|tier_i| := |\{a \in A | a \text{ is in tier } i\}|$. By combining the independent tiers we obtain $\prod_{i=0}^{7} |tier_i|! \approx 10^{1021}$ different assembly sequences. The calculation of the APG took 165 seconds, which is therefore negligible compared against the overall run time of 30 hours.

In summary, our framework ① is capable of computing an APG for our complex real-world CAD data set quickly and reliably. The experiments highlight the need for two different motion planners for the NEAR and FAR planning phase. The runtime intensive NEAR planning for parts that cannot be disassembled in the current shoring state demonstrates the importance of reducing the number of IDQ, which we did with our two algorithms VP existence and collision perceiving.

6

CONCLUSION

6.1 Summary

In the previous chapters, we tackled assembly sequence planning (ASP) for complex real-world data. The main challenge is to find and represent disassembly sequences. We explained the differences between academic examples and real-world data in Chapter 2. These are mainly: flexible (sub)parts, many more assembled parts and a large workspace that, for each part, provides different disassembly paths that differ in terms of length. The main contribution of this thesis is our ASP framework from Chapter 5 which is the first one capable of handling real-world data such as our vehicle from the automotive industry. Our framework is based on the general Voronoi diagram (GVD) from Chapter 3 and our novel *Expansive Voronoi Tree (EVT)* from Chapter 4. In the following, we quantify and discuss the contributions of each chapter.

6.1.1 General Voronoi Diagram (GVD)

We started with the definition of the ordinary Voronoi diagram. We then broadened this definition such that the resulting GVD is suitable for three-dimensional meshes. In addition, we added the medial axis (MA) to the GVD. We showed that the MA in its original definition heavily floods the GVD so we introduced a constraint that thins out the MA but still contains the important surfaces. This results in a more meaningful GVD. The main contribution of this section was our voxel-propagation algorithm which approximates the GVD. Our novel *Voronoi voxel history (VVH)* is RAM-saving but it requires more calculation time. The common three-dimensional

grid as the underlying data structure has the opposite properties; more RAM usage but faster calculation time. We tested and compared our propagation algorithm on our real-world data sets.

The second part of this chapter covered the general Voronoi diagram graph (GVDG). Extracting a GVDG from the GVD and using it as a basis for path planning is the current state-of-the-art. However, no extensive research has been conducted into how to extract a GVDG from a three-dimensional GVD. We discussed and analyzed three different approaches. The main observation is that the GVDG may consist of non-connected components. This can result in *isolated sites*. For these isolated sites, the GVDG does not provide a so called *Voronoi path (VP)* from the corresponding Voronoi cell along the GVD to a goal position. However, our presented *outside-to-inside (OTI)* propagation algorithm in combination with the *handleFalseIsolatedSites* function ensures that the GVDG provides a VP for each part that can be disassembled in the current state. This means that there is no need to search for an isolated site for a feasible disassembly path. In addition, the OTI approach delivers short VPs, which are an ideal basis for our *Expansive Voronoi Tree* motion planner. Summarized, we discussed the methods of extracting a GVDG from a GVD and with our OTI approach, presented a GVDG that is useful for the motion planning process itself and an assembly sequence planning framework.

6.1.2 Motion Planning

Chapter 4 covered the research field of motion planning. After introducing the common motion planning problem, we analyzed motion planning in the context of ASP for real-world data. Flexible (sub)parts may initially collide with their neighbors. In addition, to find a collision-free disassembly path these (sub)parts need to be deformed during the disassembly process. Simulating this physical deformation is excessively time-consuming, so we allowed a degree of collision for flexible (sub)parts. This is only necessary, however, until a part is unlocked. The subsequent requirement is to find a short disassembly path in the large workspace, so we subdivided the motion planning phase into the *NEAR*- and *FAR planning phase* (Masan, 2015). The original definition of the motion planning problem was broadened for the NEAR planning phase in such a way, that controlled and reasonable collisions are allowed. It was thus possible to reasonably unlock flexible (sub)parts. We then discussed both planning

phases separately. While we did not provide a contribution for the NEAR planning phase, we did discuss related works and chose the *Iterative Mesh Modification Planner (IMMP)* from Hegewald *et al.* (2022) as a suitable motion planner for this task.

The novel *Expansive Voronoi Tree (EVT)* motion planner, which we presented here, fulfills all the requirements of the FAR planning phase: To be reliably fast; to be capable of passing through narrow passages; and to find short disassembly paths. This motion planner uses the GVDG as a basis and samples along a VP. Roughly speaking, the EVT always knows the right forward direction which results in the aforementioned benefits. In addition, we proved that the EVT inherits the probabilistic completeness from the EST motion planner. We showed that, in the context of ASP, our EVT is superior to the state-of-the-art planners RRT, RRT*, EST and EST-connect. The runtime bottleneck of the EVT is the calculation of the GVDG. However, in the context of ASP the GVDG approximation runtime is distributed across all assembled parts and therefore negligible. Summarized, our proposed EVT motion planner is ideal in the context of ASP.

6.1.3 Assembly Sequence Planning (ASP)

The primary focus of this thesis is ASP for real-world data, which is covered in this chapter. The basis is the definition of the original ASP problem. Like in the previous chapters, we broadened the definitions such that they are suitable for real-world data. This means, that disassembly paths for the assembled parts are permitted to contain a controlled and reasonable amount of collision in the NEAR planning phase. We defined the goal of the ASP problem as follows: At least one feasible disassembly sequence must be found. The discussion of the assembly precedence graph showed that such presentations for the disassembly sequences found become excessively large for big data sets. We therefore introduced our *assembly priority graph (APG)*, which, for all assembled parts, represents the set of parts that need to be disassembled first. We proved that the size of this directed hypergraph grows quadratically only with respect to the number of assembled parts and linearly with respect to the number of disassembly paths. This means that our APG does not require much storage usage, even for a many assembled parts and many different disassembly paths for each assembled part.

Next, we presented our ASP framework. This framework heavily benefits from the previous two chapters. After elaborating the requirements for an ASP framework

capable of handling real-world data, we presented the ideas behind our framework. We used the aforementioned NEAR- and FAR planning phases for the motion planning process. We used the work from Hegewald *et al.* (2022) for the NEAR planning phase and our EVT for the FAR planning phase. This results in a motion planning process that treats flexible fastening elements and quickly finds short disassembly paths. We presented two novel heuristics to detect enclosed assembled parts. The first algorithm is based on the observations of our GVDG. If the GVDG does not provide a VP, this means that no feasible disassembly path exists. Furthermore, if a motion planning attempt was unsuccessful, we store all the parts that collided during the sampling process of the motion planners with this part. Our framework performs a renewed search for a disassembly path for this part only, when at least one of the parts in collision is disassembled. These two heuristics speed up our framework significantly.

The experiments showed that our framework is capable of solving the ASP problem for real-world data such as our vehicle from the automotive industry. Summarized, this framework is the first one that is capable of handling (*i*) flexible fastening elements; (*ii*) finds short paths; (*iii*) is fast; and (*iv*) offers a practicable way to represent the disassembly paths found.

6.2 Future Work

6.2.1 General Voronoi Diagram

Since the calculation time of the GVD is negligible in the context of ASP, our ASP framework would not benefit from being any faster. When it comes to computational geometry, however, greater speed is obviously important. As mentioned in the related work section, many attempts have been made to parallelize wavefront propagations. A parallelized divide-and-conquer algorithm could speed up the propagation algorithm by a constant factor. The algorithm works as follows: The three-dimensional grid is subdivided into disjoint subgrids. The sub-grids are calculated in parallel and then the boundaries are merged.

As shown above, due to our thinning-out parameter δ_{MA}, the GVD can consist of multiple independent components. The $handleFalseIsolatedSites$ function connects the false isolated sites such that a VP can be found for these parts. However, creating

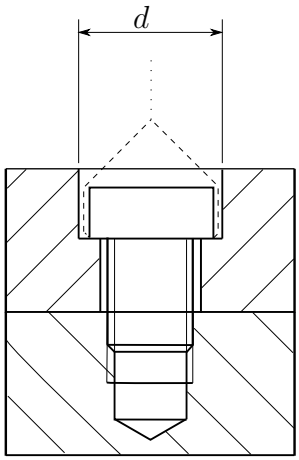

Fig. 53: A screw that fastens the two different hatched parts. The dashed line represents the GVD between the screw and the hatched part. The dotted line represents the medial axis of the hatched part.

a GVD that consists of one component and is not flooded is a more elegant and streamlined method. A potential solution that needs to be evaluated is the following. Consider the MA in Figure 53 which is represented with the dotted line. By setting $\delta_{MA} > d$ the MA is prevented. It can be allowed that a voxel becomes an MA voxel even for $\delta_{MA} > d$ if there is a GVD or MA voxel in the neighborhood of these voxels. We can therefore ensure that the MA can grow out of a Voronoi cell. However, it still needs to be determined whether there are cases where the MA can still flood the GVD.

6.2.2 Motion Planning

Motion planning is a very extensive field in which a great deal of research has been conducted. However, planning of a three-dimensional part along a GVD is little analyzed. We presented an approach whereby we extract a GVDG from the GVD and a VP from the GVDG. This assumes that a feasible disassembly path exists for the robot along the VP. We estimate this by the size of the oriented bounding box of the part and the clearance of the VP. Obviously, this estimation may not always be entirely accurate. A more generalized way of searching for a feasible disassembly path along a GVD is the following: Instead of sampling along a VP, a conventional motion planner such as the RRT or EST can be used and the sampled configurations are pushed to

the GVD. This procedure still has the advantage that the sampled configurations are positioned close to the GVD and so close to the maximum clearance in the workspace. In addition, it has the freedom to explore in all directions. However, this is offset by the fact that there is no leading direction as there is with a VP. This therefore constitutes a hybrid version between a completely unguided RRT/EST and our strongly guided EVT.

The IMMP motion planner from Hegewald *et al.* (2022) handles flexible fastening elements and delivers, in most of all cases, a reasonable disassembly path. However, this motion planner cannot handle completely flexible parts such as hoses, seals or foot mats made of fabric. The task of finding fast reasonable motion planners for completely flexible parts therefore still requires further research. An improvement for the IMMP could be the detection of flexible fastening elements with an AI approach. However, AI methods present the usual difficulties: A high volume of training data is needed and the data set needs to be labeled. How should the mesh be represented? If enough labeled data could be provided, however, an AI approach could efficiently detect flexible fastening elements or classify each part. Subsequent motion planning could then be performed with different motion planners for each class (rigid part, part with flexible fastening elements, completely flexible, etc.). For each class of parts, therefore, a special motion planner is available that addresses the various special requirements regarding different, for example, the different time limits after which a motion planner terminates.

6.2.3 Assembly Sequence Planning

Classifying each part into multiple groups (e.g. flexible part, fastening element, rigid part, completely flexible part, covering part, etc.) could offer a number of advantages. We will now present our thoughts about how such information could be applied. Assume, therefore, for the rest of this subsection that each part belongs to multiple classes such as ('flexibility = flexible', 'fastener = true', 'enclosed = false'). We subdivide the possible future work into two subgroups: calculation time and quality.

Calculation time: Solving the ASP problem for real-world data is a very time-consuming task, which is why much faster speed ups will always improve an ASP framework.

As already mentioned, motion planning accounts for most of the calculation time. Speeding up motion planning, therefore, is the best way to deliver a measurable increase in speed. On the one hand, the speed of the motion planners themselves can be improved. That said, research into motion planning has a long and well-known history and no significant increase in the speed of current motion planners can be expected. On the other hand, most motion planning attempts are made for parts that cannot be disassembled in the current state. These motion planning attempts are known as *impossible disassembly queries (IDQ)*. First, most motion planning attempts are IDQ; and second, these are even more time-consuming than attempts with a successful outcome. The reason for this is that if no disassembly path exists, the motion planners terminate only after the given maximum time limit. Therefore, reducing the number of IDQ is an easy way to significantly reduce the calculation time of an ASP framework. On the basis of the knowledge of the aforementioned part categories, a *fastener - fastened part* relationship, as it is described in Adesso *et al.* (2022), can be employed. First, check which fasteners are neighbored to which fastened parts and then disassemble the parts in the following alternating steps: Attempt to disassemble the set of fasteners and subsequently the set of fastened parts.

All parts that are in the same tier can be disassembled independently of each other allowing the motion planning attempts to be parallelized. This should significantly speed up the framework.

Quality: We will first describe how the process of finding disassembly paths can be improved, i.e. handling more parts and reducing errors in motion planning.

This thesis demonstrated that motion planning is one of the main issues in ASP. The ability to handle completely flexible parts in an adequate time is therefore crucial for a fully automated disassembly simulation. However, the problem of simulating completely flexible parts such as wires is long-standing and so an easy solution is unlikely to be found any time soon.

We assumed that the assembly can be disassembled *monotonous*, i.e. it is possible to disassemble one part after another without moving another part in between. However, this is not always the case for real-world data. For example, we excluded the doors, bonnet and tailgate because these parts often need to be opened and closed during the assembly process. In addition, non-monotone disassembly is rare but can occur. If the disassembly process terminates at a local point and a non-monotonous disassembly process is assumed we suggest the following procedure: Let $\tilde{A} \subseteq A$ be the set of local

parts where the disassembly process stopped and that are still assembled. Let $a \in \tilde{A}$ be the part to be disassembled. Now remove another part $b \in \tilde{A} \setminus \{a\}$ and make a motion planning attempt for a. If this attempt was successful, insert b again and attempt to move b such that a can pass b without collision. If the attempt was unsuccessful, change the part b that was removed. If no disassembly path can be found for a repeat this procedure for another part of $\tilde{A}$. This procedure needs quadratically many disassembly attempts depending on $|\tilde{A}|$.

The ability to detect subassemblies is useful. If multiple parts can be grouped into one subassembly, the number of motion planning attempts required is reduced. Moreover, subassemblies are of great practical interest. Assembling a subassembly can be performed parallel to the main assembly process, which helps to speed up the overall assembly process.

To find a good VP, we used only the length of a path as a criterion. Disassembly paths can also be evaluated in terms of ergonomic.

Our assembly priority graph is an excellent method of representing the disassembly paths found. However, the APG does not evaluate the different disassembly sequences. The process of evaluating a disassembly sequence and subsequently finding the optimal one is still a task that requires further investigation.

OWN REFERENCES

Dorn, S., N. Wolpert, and E. Schömer (2020). "Voxel-based General Voronoi Diagram for Complex Data with Application on Motion Planning". In: *International Conference on Robotics and Automation (ICRA)*, pp. 137–143.

Dorn, S., N. Wolpert, and E. Schömer (2021). "Expansive Voronoi Tree: A Motion Planner for Assembly Sequence Planning". In: *International Conference on Robotics and Automation (ICRA)*, pp. 7880–7886.

Dorn, S., N. Wolpert, and E. Schömer (2022). "An Assembly Sequence Planning Framework for Complex Data using General Voronoi Diagram". In: *accepted at: International Conference on Robotics and Automation (ICRA)*.

REFERENCES

Adesso, M. F., R. Hegewald, N. Wolpert, E. Schömer, B. Maier, and B. A. Epple (2022). "Automatic Classification and Disassembly of Fasteners in Industrial 3D CAD-Scenarios". In: *accepted at: International Conference on Robotics and Automation (ICRA)*.

Aguinaga, I., D. Borro, and L. Matey (2008). "Parallel RRT-based path planning for selective disassembly planning". In: *International Journal of Advanced Manufacturing Technology* 36, pp. 1221–1233.

Ausiello, G. and L. Laura (2017). "Directed hypergraphs: Introduction and fundamental algorithms - A survey". In: *Theoretical Computer Science*, pp. 293–306.

Baston, I. and N. Celes (2008). "Approximation of 2d and 3d generalized Voronoi Diagrams". In: *International Journal of Computer Mathematics*, pp. 1003–1022.

Boissonnat, J.-D. and M. Teillaud (2006). *Effective Computational Geometry for Curves and Surfaces (Mathematics and Visualization)*. Berlin, Heidelberg: Springer-Verlag, pp. 67–116. ISBN: 3540332588.

Cao, T.-T., K. Tang, A. Mohamed, and T.-S. Tan (2010). "Parallel banding algorithm to compute exact distance transform with the GPU". In: *SIGGRAPH*, pp. 83–90.

Danielsson, P.-E. (1980). "Euclidean distance mapping". In: *Computer Graphics and Image Processing*, pp. 227–248.

Denny, J., E. Greco, S. Thomas, and N. M. Amato (2014). "MARRT: Medial Axis biased rapidly-exploring random trees". In: *International Conference on Robotics and Automation (ICRA)*, pp. 90–97.

Descartes, R. (1644). "Principia Philosophiae". In: *Ludovicus Elzevirius*.

Ebinger, T., S. Kaden, S. Thomas, R. Andre, N. M. Amato, and U. Thomas (2018). "A general and flexible search framework for disassembly planning". In: *International Conference on Robotics and Automation (ICRA)*.

Edwards, J., E. Daniel, V. Pascucci, and C. Bajaj (2015). "Approximating the Generalized Voronoi Diagram of closely spaced objects". In: *EUROGRAPHICS* 34.

Fang, S. and H. Chen (2000). "Hardware accelerated voxelization". In: *Springer Tracts in Advanced Robotics*, pp. 433–442.

Fortune, S. (1987). "A sweepline algorithm for Voronoi diagrams". In: *Algorithmica* 2.

Foskey, M., M. Garber, M. C. Lin, and D. Manocha (2001). "A Voronoi-based hybrid motion planner". In: *International Conference on Intelligent Robots and Systems*.

Garber, M. and M. C. Lin (2002). "Constraint-based motion planning using Voronoi Diagrams". In: *International Conference on Robotics and Automation (ICRA)*, pp. 541–558.

Geraerts, R. (2010). "Planning short paths with clearance using explicit corridors". In: *International Conference on Robotics and Automation (ICRA)*, pp. 1997–2004.

Ghandi, S. and E. Masehian (2015). "Review and taxonomies of assembly and disassembly path planning problems and approaches". In: *Journal of Computer-Aided Design* 67-68, pp. 58–86.

Gottschalk, S., L. C. Ming, and D. Manocha (1996). "OBBTree: A Hierarchical Structure for Rapid Interference Detection". In: *SIGGRAPH '96*, pp. 171–180.

Guo, L., F. Wang, Z. Huang, and N. Gu (2011). "A fast and robust seed flooding algorithm on GPU for Voronoi Diagram generation". In: *International Conference on Electrical and Control Engineering*, pp. 492–495.

Hadj, R. B., I. Belhadj, M. Trigui, and N. Aifaoui (2018). "Assembly sequences plan generation using features simplification". In: *Journal of Advances in Engineering Software* 119, 1–11.

Hegewald, R., N. Wolpert, and E. Schömer (2021). "Saliency Features for 3D CAD-Data in the Context of Sampling-Based Motion Planning". In: *International Conference on Robotics and Automation (ICRA)*, pp. 7858–7864.

Hegewald, R., N. Wolpert, and E. Schömer (2022). "Iterative Mesh Modification Planning: A new Method for Automatic Disassembly Planning of Complex Industrial Components". In: *accepted at: International Conference on Robotics and Automation (ICRA)*.

Hoff, K., T. Culver, J. Keyser, M. C. Lin, and D. Manocha (2000). "Interactive motion planning using hardware-accelerated computation of generalized Voronoi diagrams". In: *International Conference on Robotics and Automation (ICRA)*, pp. 2931–2937.

Hoff, K. E., J. Keyser, M. Lin, D. Manocha, and T. Culver (1999). "Fast computation of generalized Voronoi Diagrams using graphics hardware". In: *SIGGRAPH*, pp. 227–286.

Holleman, C. and L. E. Kavraki (2000). "A framework for using the workspace medial axis in prm planners". In: *International Conference on Robotics and Automation (ICRA)* 2, 1408–1413.

Hsieh, H. and W. Tai (2005). "A simple GPU-based approach for 3D Voronoi diagram construction and visualization". In: *Simulation Modelling Pratice and Theory*, pp. 681–692.

Hsu, D., J.-C. Latombe, and R. Motwani (1997). "Path planning in expansive configuration spaces". In: *International Conference on Robotics and Automation (ICRA)* 3, 2719–2726.

Jiménez, P. (2013). "Survey on assembly sequencing: a combinatorial and geometrical perspective". In: *Journal of Intelligent Manufacturing*, pp. 235–250.

Jones, M. W., J. A. Bærentzen, and M. Sramek (2006). "3D distance fields: a survey of techniques and applications". In: *IEEE Transactions on Visualization on Computer Graphics* 12, pp. 581–599.

Karaman, S. and E. Frazzoli (2011). "Sampling-based algorithms for optimal motion planning". In: *International Journal of Robotics Research (IJRR)* 30, pp. 846–894.

Kardos, C. and J. Váncza (2017). "Application of Generic CAD Models for Supporting Feature Based Assembly Process Planning". In: *Conference on Intelligent Computation in Manufacturing Engineering (CIRP)*.

Kavraki, L., J.-C. Latombe, and R. H. Wilson (1993). "On the complexity of assembly partitioning". In: *Information Processing Letters* 48, pp. 229–235.

Kavraki, L., P. Svestka, J. C. Latombe, and M. H. Overmars (1996). "Probabilistic Roadmaps for Path Planning in High-Dimensional Configuration Spaces". In: *IEEE Transactions on Robotics and Automation* 12, 566–580.

Kuffner, J. J. and S. M. LaValle (2000). "RRT-connect: An efficient approach to single-query path planning". In: *International Conference on Robotics and Automation (ICRA)* 2, pp. 995–1001.

Lavalle, S. M. (1998). "Rapidly-Exploring Random Trees: A New Tool for Path Planning". In: *Tech. Rep.*

Lavender, D., A. Bowyer, J. Davenport, A. Wallis, and J. Woodwark (1992). "Voronoi Diagrams of set-theoretic solid models". In: *IEEE Comput. Graph*, pp. 69–77.

Masan, G. (2015). *personal communication under:* `gerald.masan@daimler.com`.

Masehian, E. and S. Gahndi (2021). "Assembly sequence and path planning for monotone and nonmonotone assemblies with rigid and flexible parts". In: *Robotics and Computer-Integrated Manufacturing* 72.

Mello, L. S. Homem de and A. C. Sanderson (1988). "Automatic generation of mechanical assembly sequences". In: *Carnegie-Mellon University Pittsburgh PA Robotics Institute, Tech. Rep.*

Michniewicz, J. J. (2019). "Automatische simulationsgestützte Arbeitsplanung in der Montage". PhD thesis. Technische Universität München.

Müller, M., B. Heidelberger, M. Hennix, and J. Ratcliff (2007). "Position Based Dynamics". In: *Journal of Visual Communication and Image Representation*, pp. 109–118.

Okabe, A., B. Boots, K. Sugihara, and S. Chiu (Jan. 2000). *Spatial Tessellations: Concepts and Applications of Voronoi Diagrams*. Vol. 43. ISBN: 9780471986355.

Ou, L. and X. Xui (2013). "Relationship matrix based automatic assembly sequence generation from a CAD model". In: *Journal of Computer-Aided Design* 45, 1053–1067.

Pintzos, G., C. Triantafyllou, N. Papakostas, D. Mourtzis, and G. Chryssolouris (2016). "Assembly precedence diagram generation through assembly tiers determination". In: *International Journal of Advanced Manufacturing Technology*, pp. 1045–1057.

Popescu, D. and R. Iacob (2013). "Disassembly method based on connection interface". In: *Journal of Advanced Manufacturing Technology* 69, pp. 1511–1525.

Ragnemalm, I. (1992). "Fast erosion and dilation by contour processing and thresholding of distance maps". In: *Pettern Recognition Letters* 13, pp. 161–166.

Rong, G. and T. Tan (2006). "Jump Flooding in GPU with applications to Voronoi Diagram and distance transform". In: *Symposium of Interactive 3D graphics and games*, pp. 109–116.

Schneider, D. (2017). "Rigid and Deformable Motion and Disassembly Planning". PhD thesis. Johannes Gutenberg-University Mainz.

Schneider, J., M. Kraus, and R. Westermann (2009). "GPU-based real-time discrete euclidean distance transforms with precise error bounds". In: *VISAPP*, pp. 435–440.

Seda, M. (2007). "Roadmap Methods vs. Cell Decomposition in Robot Motion Planning". In: *International Conference on Signal Processing, Robotics and Automation*, pp. 127–132.

Seda, M. and V. Pich (2008). "Robot Motion Planning Using Generalised Voronoi Diagrams". In: *International Conference on Signal Processing, Robotics and Automation*, pp. 215–220.

Strzodka, R. and A. Telea (2004). "Generalized distance transforms and skeletons in graphics hardware". In: *IEEE TCVG Symposium on Visualization*, pp. 221–230.

Su, Q. (2006). "Copmuter aided geometric feasible assembly sequence planning and optimizing". In: *International Journal of Advanced Manufacturing Technology* 33, 48–57.

Tsitsiklis, J. N. (1995). "Efficient Algorithms for Globally Optimal Trajectories". In: *IEEE Transactions on Automatic Control* 40.9, pp. 1528–1538.

Velic, M., D. May, and L. Moresi (2009). "A fast robust algorithm for computing discrete Voronoi Diagrams". In: *Journal of Mathematical Modelling and Algorithms*, pp. 343–355.

Wang, X., C. Yang, J. Wang, and X. Meng (2011). "Hierarchical Voronoi diagram-based path planning among polygonal obstacles for 3D virtual worlds". In: *International Symposium on VR Innovation*, pp. 175–181.

Wilmarth, S. A., N. M. Amato, and P. F. Stiller (1999). "MAPRM: A probabilistic roadmap planner with sampling on the medial axis of the free space". In: *International Conference on Robotics and Automation (ICRA)* 2, 1024–1031.

Wilson, R. H. (1992). “On geometric assembly planning”. PhD thesis. Stanford university.

Yang, L., J. Qi, D. Song, J. Xiao, J. Han, and Y. Xia (2016). “Survey of Robot 3D Path Planning Algorithms”. In: *Journal of Control Science and Engineering*.

Yu, J. and C. Wang (2013). “Method for discriminating geometric feasibility in assembly planning based on extended and turning interference matrix”. In: *Journal of Advanced Manufacturing Technology* 67, 1867–1882.

Yuan, Z., G. Rong, X. Guo, and W. Wang (2011). “Generalized Voronoi Diagram computation on GPU”. In: *International Symposium on Voronoi Diagrams in Science and Engineering*.

Zhang, W., M. Ma, and H. Li (2017). “Generating interference matrices for automatic assembly sequence planning”. In: *International Conference on Methods and Models in Automation and Robotics (MMAR)* 90, 1187–1201.

Lebenslauf

Sebastian Dorn

Geburt 01. Juli 1989 in Wertheim, deutsch

Wohnort Obere Waiblinger Straße 126, 70374 Stuttgart

Kontakt ✉ sebastian.dorn89@gmx.de, ☎ 0171/3801418

Ausbildung

2018–jetzt **Dissertation an der Johannes Gutenberg-Universität Mainz**,
Unter der Betreuung von Prof. Dr. Elmar Schömer und Prof. Dr. Nicola Wolpert,
Industriepartner: Mercedes-Benz AG,
Thema: Assembly Sequence Planning for Complex real-world Data.
Entwicklung eines Assembly Sequence Planning Frameworks für 3D CAD Daten: 3D General Voronoi Diagram Approximation, Entwicklung eines optimierten Motion Planners.
Tools: C++, QT, OpenGL, CMake.

2016–2018 **Master of Science, Mathematik**, *Julius-Maximilians-Universität Würzburg*, ∅ 1,7, Nebenfach: Informatik, Abschlussarbeit: Approximationsalgorithmen für distanzbeschränkte Standortprobleme.

2012–2016 **Bachelor of Science, Mathematik**, *Hochschule für Technik Stuttgart*, ∅ 1,5, Nebenfach: Informatik, Abschlussarbeit: Beleuchtungsoptimierung mittels einer Kontrastbewertung für industrielle Sichtprüfungen.

2010–2012 **Staatlich geprüfter Techniker**, *Gewerbliche Schule Tauberbischofsheim*, ∅ 1,4.

2005–2009 **Technischer Zeichner**, *kurtz ersa GmbH Kreuzwertheim*, ∅ 1,5.

2005 **Mittlere Reife**, *Comenius Realschule Wertheim*, ∅ 2,5.

Berufliche Tätigkeit

2022–jetzt **Mercedes-Benz AG**, *Böblingen*, Algorithmenentwickler.

2018–2022 **Mercedes-Benz AG**, *Sindelfingen*, Doktorand.

2015–2017 **IT-Designers GmbH**, *Esslingen*, Werkstudent, Pixel-Weltkoordinaten Transformation, Bestimmung der Kameraposition, Partikelfilter zur Objektverfolgung.
Tools: C#, Scala.

2014 **IT-Designers GmbH**, *Esslingen*, Praktikant, Visual Data Framework. Tools: C#, R.

2009–2010 **kurtz ersa GmbH**, *Wertheim*, Technischer Zeichner.

Veröffentlichungen

2022 **An Assembly Sequence Planning Framework for Complex Data using General Voronoi Diagram**, *International Conference on Robotics and Automation (ICRA)*.

2021 **Expansive Voronoi Tree: A Motion Planner for Assembly Sequence Planning**, *International Conference on Robotics and Automation (ICRA)*.

2020 **Voxel-based General Voronoi Diagram for Complex Data with Application on Motion Planning**, *International Conference on Robotics and Automation (ICRA)*.