Assembly Sequence Planning Using Deep Learning Approaches

Zur Erlangung des akademischen Grades eines

Doktors der Ingenieurwissenschaften

von der KIT-Fakultät für Informatik
des Karlsruher Instituts für Technologie (KIT)

genehmigte

Dissertation

von

Alexander Cebulla

Tag der mündlichen Prüfung: 6. Dezember 2024

1. Referent: Prof. Dr.-Ing. Tamim Asfour
2. Referent: Dr.-Ing. Torsten Kröger

I am deeply grateful to my family and my girlfriend for their unconditional love and support, to my friends for their encouragement and understanding, and to my professors and colleagues for their invaluable guidance and collaboration.
This work would not have been possible without you all.

Abstract

Assembly Sequence Planning Using Deep Learning Approaches

One of the visions of Industry 4.0 is to enable mass customization through flexible production systems. A key challenge in realizing this vision is efficiently planning assembly sequences for customized product variants. Traditionally, assembly sequence planning (ASP) has been performed manually by experts, but this process is time-consuming and does not scale well. This work contributes to the automation and optimization of ASP.

One approach to automate ASP is through Assembly-by-Disassembly (AbD) simulation. This method starts with the fully assembled product and iteratively removes parts one by one. After each part is removed, the system performs two checks: first, it tests whether the part could be removed without collisions; second, it verifies that the remaining assembly is stable. This continues until all parts have been removed. Inverting the disassembly sequence then provides an assembly sequence. This inversion is possible if no irreversible joining methods, such as welding or gluing, are used. While general, AbD faces a combinatorial explosion and may require up to N^2 tests for an assembly with N parts. To improve the efficiency of feasible ASP, we propose a graph-based assembly representation and use a Graph Neural Network (GNN) to predict part removability. We then incorporate these predictions into a graph search algorithm that prioritizes testing parts with high predicted removability. With this approach, we could significantly reduce the number of unsuccessful removal tests compared to traditional AbD approaches.

Manufacturers often require not only feasible sequences but also want them to be optimized for specific objectives. However, searching for optimal assembly sequences faces an even greater combinatorial challenge, i. e., in the worst case, $N!$ sequences must be evaluated. We address single-objective optimization with a framework that combines Monte Carlo Tree Search (MCTS) with Deep Q-learning (DQL). When evaluated on datasets of aluminum profile assemblies, this approach consistently outperformed vanilla MCTS in minimizing total removal path length.

Additionally, manufacturers often want to optimize multiple objectives simultaneously. Typical approaches employ linear combinations of objectives, but this provides less insight into the trade-offs between different objectives and may overlook potentially valuable solutions. To address these limitations, we extend our framework to multi-objective optimization. Specifically, we use multiple Q-functions to guide a multi-objective variant of MCTS towards Pareto-optimal assembly sequences. This provides manufacturers with a set of non-dominated solutions that offer clear insights into trade-offs. Experiments on Soma cube assemblies demonstrated the effectiveness of our approach in simultaneously optimizing geometric accessibility and minimizing direction changes.

A significant challenge to conducting this research is the availability of suitable datasets. Existing mechanical assembly datasets often lack detailed information about their assemblies, i. e., what tools are required or how many robotic manipulators are needed. In contrast, assembly benchmarks provide this information but have limited variety. To address this, we present two assembly generators: one for 3D aluminum profile assemblies that can be assembled by a single robotic manipulator and another for Soma cubes of arbitrary size.

Zusammenfassung

Planung Von Montagesequenzen Mittels Deep Learning Ansätzen

Eine der Visionen von Industrie 4.0 ist, kundenindividuelle Massenproduktion durch flexible Produktionssysteme zu ermöglichen. Eine zentrale Herausforderung bei der Verwirklichung dieser Vision ist die effiziente Planung von Montagesequenzen für kundenspezifische Produktvarianten. Traditionell wurde die Planung von Montagesequenzen von Experten manuell durchgeführt, was zeitaufwendig und schlecht skalierbar ist. Diese Arbeit trägt zur Automatisierung und Optimierung der Montagesequenzplanung bei.

Ein Ansatz zur Automatisierung der Planung von Montagesequenzen ist die AbD-Simulation. Diese Methode beginnt mit dem vollständig montierten Produkt und entfernt iterativ einen Teil nach dem anderen. Nach jeder Entfernung führt das System zwei Überprüfungen durch: Zunächst wird geprüft, ob das Teil kollisionsfrei entfernt werden konnte; anschließend wird überprüft, ob die verbleibende Baugruppe stabil ist. Dies wird fortgesetzt, bis alle Teile entfernt sind. Durch die Umkehrung der Demontagesequenz ergibt sich dann eine Montagesequenz. Diese Umkehrung ist möglich, wenn keine irreversiblen Verbindungstechniken wie Schweißen oder Kleben eingesetzt werden. Obwohl allgemeingültig, steht die AbD-Methode vor einer kombinatorischen Explosion und kann für eine Baugruppe mit N Teilen bis zu N^2 Tests erfordern.

Um die Effizienz der Planung durchführbarer Montagesequenzen zu verbessern, schlagen wir eine graphbasierte Darstellung der Baugruppe vor und nutzen ein GNN, um die Entfernbarkeit der Teile vorherzusagen. Diese Vorhersagen werden dann in einen Graphsuchalgorithmus integriert, der das Testen von Teilen mit hoher vorhergesagter Entfernbarkeit priorisiert. Mit diesem Ansatz konnten wir die Anzahl der fehlgeschlagenen Entfernungstests im Vergleich zu traditionellen AbD-Ansätzen deutlich reduzieren.

Hersteller benötigen oft nicht nur durchführbare Sequenzen, sondern wollen diese auch für spezifische Ziele optimiert haben. Die Suche nach optimalen Montagesequenzen steht jedoch vor einer noch größeren kombinatorischen Herausforderung, d.h. im schlimmsten Fall müssen $N!$ Sequenzen bewertet werden. Für diese Einzelzieloptimierung schlagen wir ein Framework vor, das MCTS mit DQL kombiniert. Bei der Evaluierung auf Datensätzen von Aluminiumprofil-Baugruppen übertraf dieser Ansatz konsequent reines MCTS bei der Minimierung der Gesamtlänge des Entfernungspfads.

Darüber hinaus wollen Hersteller oft mehrere Ziele gleichzeitig optimieren. Typische Ansätze verwenden lineare Kombinationen von Zielen, was jedoch weniger Einblicke in die Kompromisse zwischen verschiedenen Zielen bietet und potenziell wertvolle Lösungen übersehen kann. Um diesen Einschränkungen zu begegnen, erweitern wir unser Framework auf Mehrzieloptimierung. Konkret verwenden wir mehrere Q-Funktionen, um eine

mehrzielorientierte Variante von MCTS zu Pareto-optimalen Montagesequenzen zu führen. Dies bietet Herstellern eine Menge nicht-dominierter Lösungen, die klare Einblicke in die Kompromisse zwischen den verschiedenen Zielen ermöglichen. Experimente mit Soma-Würfeln zeigen die Effektivität des Ansatzes bei der gleichzeitigen Maximierung der geometrischen Zugänglichkeit und der Minimierung von Richtungswechseln.

Eine bedeutende Herausforderung bei der Durchführung dieser Forschung ist die Verfügbarkeit geeigneter Datensätze. Bestehende Datensätze für mechanische Baugruppen enthalten oft keine detaillierten Informationen über ihre Montage, z.B. welche Werkzeuge erforderlich sind oder wie viele Roboter-Manipulatoren benötigt werden. Im Gegensatz dazu bieten Montage-Benchmarks diese Informationen, weisen jedoch oft nur eine begrenzte Vielfalt auf. Um dies anzugehen, werden zwei Baugruppen-Generatoren entwickelt: einer für 3D-Aluminiumprofil-Baugruppen, die von einem einzelnen Roboter-Manipulator montiert werden können, und ein weiterer für Soma-Würfel beliebiger Größe.

Contents

1 Introduction

1.1 Motivation

Industry 4.0 represents a vision for the future of manufacturing driven by technological advances across various fields, including additive manufacturing, sensors, robotics, and artificial intelligence. It promises smart factories characterized by interconnected, flexible, and autonomous production systems. Through constant monitoring and analysis of the production process, these systems will be able to predict equipment failures before they occur, significantly minimizing downtime. Furthermore, they will enable manufacturers to economically produce products that are tailored to the individual specifications of each customer.

However, making this a reality is an ongoing process. While companies and researchers are making major strides, achieving the full vision of Industry 4.0 still requires overcoming significant technological challenges. One of these is automating assembly sequence planning (ASP), which involves determining the sequence of operations to assemble a desired customized product from its individual parts. Often, in addition to being feasible, i.e., executable by production systems, these sequences should also be optimal with respect to certain objectives, such as minimizing tool changes or assembly path lengths.

Traditionally, assembly sequences are planned by domain experts who are familiar with the product and production systems. However, this process is time-consuming, expensive, and does not scale. One method for automating it is Assembly-by-Disassembly (AbD). Starting from the 3D model of a product in its fully assembled state, it iterates over all parts and tries to remove them in simulation. If a part can be removed, this process is repeated for the remaining parts until all are removed. The found disassembly sequence is then inverted to obtain an assembly sequence. AbD relies on the assumption that assembly and disassembly are the inverse of each other, which holds as long as all disassembly steps are non-destructive.

While AbD is a general approach for automating ASP, it faces the major challenge of combinatorial explosion in the search space. The goal of this thesis is to alleviate this problem. Specifically, we propose an ASP framework that uses deep learning techniques to focus the planning process on the most promising parts and assembly sequences. By learning from previous planning experiences, our approach can significantly reduce the computational effort required to find feasible and optimal assembly sequences.

1.2 Research Questions

To develop our novel ASP framework, the following research questions must be addressed.

1. **How to efficiently discover a feasible assembly sequence?**

   AbD iteratively attempts to remove parts from an assembly without causing collisions or compromising the stability of the remaining structure. In the worst case, for an assembly consisting of N parts, AbD may have to perform up to $\frac{N \cdot (N-1)}{2} \in \mathcal{O}(N^2)$ removal tests. While previous work has often focused on speeding up these tests, our research addresses the question of how to reduce the number of failed removal attempts by leveraging the knowledge learned from previous ASP experiences.

2. **How to efficiently discover optimal assembly sequences?**

   Manufacturers often require assembly sequences that optimize additional objectives beyond simple feasibility. This requirement significantly increases the complexity of the problem, as the search needs to consider all feasible sequences. In the worst case, an assembly with N parts has $N!$ possible sequences.

   Additionally, there may be multiple objectives to optimize simultaneously, turning this into a multi-objective optimization problem. In this case, a planning algorithm needs to find a set of assembly sequences known as the Pareto front, where each sequence is optimal for a subset of objectives.

   A range of such algorithms has been suggested in related work. However, all of them were focused on planning optimal assembly sequences for a specific assembly and had to start the planning process from scratch for each new assembly. In contrast, our approach aims to transfer knowledge from previous planning attempts to improve the efficiency of finding optimal sequences for new assemblies.

3. **How to address the lack of high-quality assembly datasets?**

   A major limiting factor in applying data-driven approaches such as deep learning to ASP is the lack of high-quality datasets of industrial assemblies. Publicly available 3D models are often incomplete, e. g., missing screws. Additionally, they may contain errors such as overlapping parts. Another problem is that the available models might require a diverse set of tools and a varying number of robots to be successfully assembled.

   Consequently, creating a comprehensive and reliable dataset for ASP becomes a highly time-consuming process, requiring researchers to invest significant effort in cleaning, correcting, and annotating the 3D models. As a result, related work often focuses on toy examples such as assembling LEGO models or wooden blocks. To address this challenge, we present a method for automatically generating industrial-grade assemblies that can be assembled by a single robot.

1.3 Contributions

To address these questions, we make the following contributions:

1. **An approach for identifying assembly parts that have a high likelihood of being removable.**

   To address the first research question, we propose a method that reduces the number of unsuccessful removal attempts by prioritizing parts that are more likely to be removable. We observe that a part's removability heavily depends on the surrounding parts. Moreover, we assume that similar parts with similar surrounding parts have a comparable likelihood of being removable. Therefore, we can learn from previous planning attempts for other assemblies whether a part could be removed in the presence of specific surrounding parts and then apply this learned knowledge during ASP for a new assembly.

   We introduce a graph-based assembly representation, where parts are represented as nodes and contacts between parts are indicated by edges. By considering the graph structure around each node, we can effectively capture the influence of surrounding parts on the removability of a given part.

   This graph representation is then used to train a Graph Neural Network (GNN), a specialized neural network architecture capable of processing graphs of varying sizes. We utilize the trained GNN to predict the removability of parts during the planning of an assembly sequence for a new assembly. Unlike related work that directly predicts an assembly sequence using deep learning, our approach uses the predictions in combination with a graph search. This guides the search towards parts with a higher likelihood of being removable, significantly reducing the number of unsuccessful removal tests and improving the efficiency of the ASP process.

2. **A strategy for the efficient discovery of an optimal assembly sequence that combines Monte Carlo Tree Search (MCTS) with deep learning.**

   To address the second research question, we propose a strategy that efficiently discovers optimal assembly sequences by combining MCTS with deep learning. This combination has been successfully applied in other domains, such as playing board games like Go and chess [101], where it has achieved superhuman performance. However, its application to efficiently finding optimal assembly sequences is novel.

   While the previous contribution focused on identifying removable parts to find a feasible assembly sequence, planning an optimal assembly sequence is a more complex challenge. In contrast to checking for feasibility, which is a binary decision that can be made at each planning step, the influence of a decision on the overall optimality of the sequence might only become apparent later in the planning process.

   We employ MCTS as a planning algorithm as it is well-suited for this kind of problem. It iteratively builds a search tree by simulating different assembly sequences and evaluating their optimality. Each node in the search tree represents a partially assembled state, and the edges represent the action of removing a specific part. By repeatedly sampling assembly sequences and propagating the results back through the tree, MCTS gradually learns which parts are more likely to lead to optimal sequences when removed at a given state.

Reinforcement Learning (RL) is another approach for solving this problem: an agent interacts with an environment by taking actions and receiving rewards to learn a policy that maximizes the cumulative reward over time. In the context of ASP, the environment is the partially assembled state, the actions are the removal of specific parts, and the rewards are based on the optimality of the resulting sequences. To learn a policy, we use Deep Q-learning (DQL), a RL algorithm that utilizes neural networks to approximate the action-value function, also known as the Q-function. In our case, the Q-function computes the expected cumulative reward of removing a specific part given the current partially assembled state. We use GNNs in combination with a graph-based assembly representation to learn this function.

We incorporate the learned Q-function into the MCTS by using it to guide the simulation step. Usually, during this step, parts are selected randomly. However, in our approach, we instead use an ϵ-greedy policy based on the Q-function to choose the part to remove. That is, with probability $1 - \epsilon$, the policy selects the part with the highest Q-value, i. e., the most promising part to remove based on the expected cumulative reward. Otherwise, a random part is removed. This allows the search to balance between exploiting the knowledge captured by the Q-function and exploring potentially promising actions that the Q-function might not have accurately estimated yet.

We extend the approach to multi-objective ASP by utilizing an extension of MCTS, where each node of the search tree constructs an approximation of a local Pareto front. The root node will therefore approximate the global Pareto front for the complete assembly sequences. Furthermore, we train one Q-function per objective. To guide the simulation step, we again employ an ϵ-greedy policy, where with probability $1 - \epsilon$, the policy selects the part that maximizes the product of the Q-values across all objectives, effectively considering the trade-offs between the objectives. Otherwise, a random part is chosen to encourage exploration.

3. **Generators for Aluminum Profile Assemblies and Soma Cubes.**

   To overcome the lack of high-quality datasets for training and evaluating our methods, we develop two assembly generators. The first generator creates assemblies out of aluminum profiles. While previous approaches have been used to generate 2D assemblies consisting of a small number (< 10) of profiles, our approach can generate stable 3D assemblies of arbitrary size that a single robot manipulator can assemble. The second generator creates Soma cubes, a type of 3D puzzle, of arbitrary size. They have the important property that all parts can be removed along straight trajectories.

1.4 Outline

Introduction
In Chapter 1, we motivate the importance of automating ASP in realizing Industry 4.0 and discuss the challenges faced by traditional ASP approaches. We then present our research questions and outline the contributions we aim to make in addressing these challenges.

State of the Art
Chapter 2 provides a comprehensive overview of current research in ASP. We review fundamental concepts, methods for generating feasible assembly sequences, approaches to optimizing assembly sequences, and existing assembly datasets and benchmarks. We also discuss the limitations of current approaches, particularly in handling the combinatorial explosion that occurs when searching for feasible and optimal assembly sequences.

Learning Feasible Assembly Actions
In Chapter 3, we present our novel approach to plan feasible assembly sequences efficiently. We introduce a graph-based assembly representation and describe how we use a GNN to predict part removability. We demonstrate how these predictions can guide a graph search to efficiently explore the disassembly graph.

Generating Assemblies
Chapter 4 introduces two assembly generators we developed to address the lack of suitable datasets for training and evaluating ASP algorithms: The first generator creates 3D aluminum profile assemblies that a single robotic manipulator can assemble, while the second generates Soma cubes, a type of 3D puzzle, of various sizes. These cubes have the property that each part can be removed along a straight line.

Learning Assembly Actions to Optimize a Single Objective
In Chapter 5, we extend our feasible ASP framework to optimize assembly sequences with respect to a single objective efficiently. We describe our method, which combines MCTS with DQL, utilizing a GNN to learn the Q-function. We evaluate this approach on datasets of aluminum profile assemblies and demonstrate its superiority over vanilla MCTS.

Learning Assembly Actions to Optimize Multiple Objectives
Chapter 6 presents our approach to multi-objective optimization in ASP. We describe how we adapt MCTS to handle Pareto optimization and explain the integration of multiple Q-functions to guide the search process. We evaluate this method on Soma cube assemblies, where we simultaneously maximize part accessibility and minimize the number of direction changes.

Discussion, Outlook and Conclusion
In Chapter 7, we summarize our contributions to automating feasible and optimal ASP. We then outline directions for future research in this field. Finally, we give some concluding remarks.

2 State of the Art

The goal of this thesis is to increase the efficiency of ASP by reusing knowledge gained during prior planning attempts. In this chapter, we provide an overview of the related literature in the field of ASP and position the work within this context.

In Section 2.1, we introduce fundamental concepts related to ASP, including different approaches to representing assemblies and their corresponding assembly sequences. We also discuss various properties of these representations.

Section 2.2 focuses on methods for generating feasible assembly sequences. We begin by presenting approaches that obtain sequences from human experts, either through direct questioning or by Learning from Demonstration (LfD). Next, we explore case-based reasoning (CBR) methods that reuse solutions from similar past assemblies. We then examine techniques that simulate the assembly process using Computer-Aided Design (CAD) models, discussing both methods that precompute blocking relationships and those that utilize sampling-based path planning. Finally, we review approaches that leverage knowledge from previous planning attempts to predict feasible assembly actions, highlighting their potential benefits and limitations.

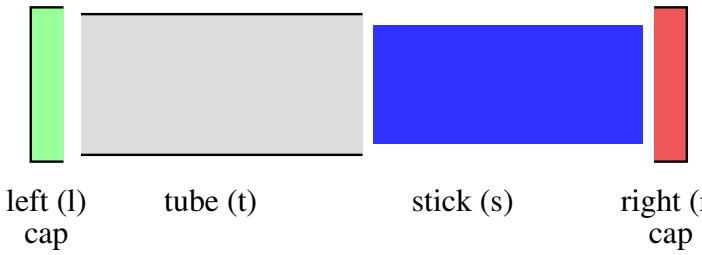
In Section 2.3, we review techniques for generating optimal assembly sequences. We first discuss exhaustive search methods, followed by soft computing approaches, such as Genetic Algorithm (GA) and Reinforcement Learning (RL). We then explore methods that transfer knowledge from previous planning efforts to optimize sequences for new assemblies more efficiently.

Finally, in Section 2.4, we provide an overview of existing datasets and benchmarks for assembly planning, describing their characteristics and limitations.

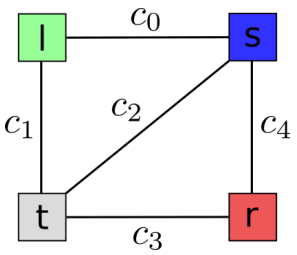
2.1 Fundamental Concepts of ASP

The ASP process takes a representation of an assembly as input and generates one or more feasible assembly sequences as output.

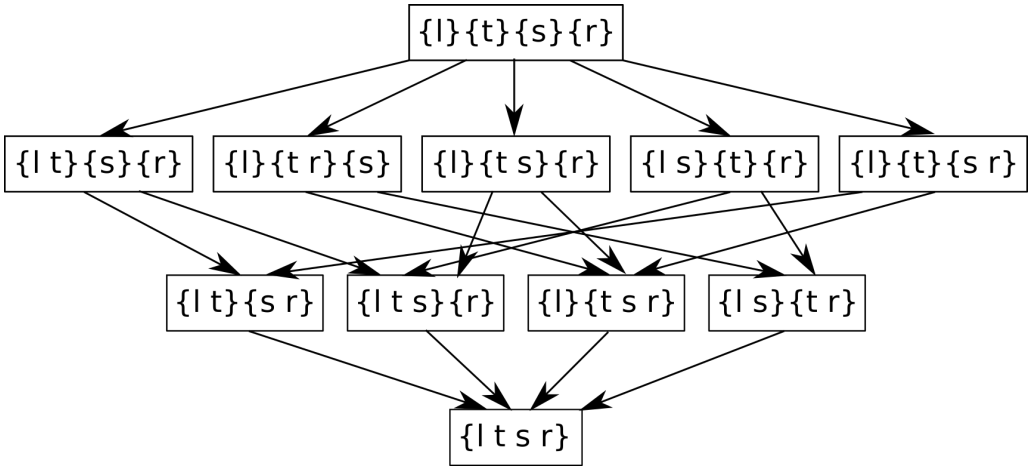
In the following sections, we will first discuss different representations of assemblies and then examine how assembly sequences can be represented. This is followed by an analysis of the conditions that determine the feasibility of these sequences. Finally, we provide an overview of assembly sequence properties, which offer insights into the “complexity” of planning and executing different sequences.



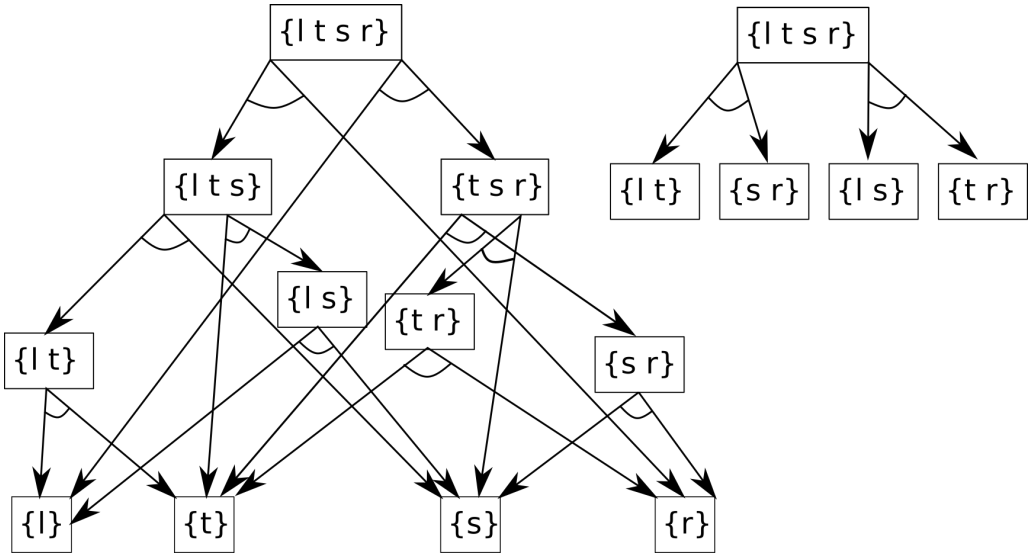
(a) A tube assembly consisting of four parts.



(b) Liaison diagram or graph of connections.



(c) directed acyclic graph (DAG).



(d) AND/OR graph (shown in two parts to improve readability).

Figure 2.1: Various representations of the tube assembly shown in (a) and its liaison diagram (b). (c) and (d) depict all of its assembly sequences as a directed acyclic graph DAG and an AND/OR graph, respectively (adapted from Homem de Mello and Sanderson [53]. © 1991, IEEE).

2.1.1 Representation of Assemblies

At its core an assembly is a tuple $\mathcal{A} := (\mathcal{P}, \mathcal{C})$, where $\mathcal{P} := \{p_0, p_1, \dots p_{N-1}\}$ is a set of N individual parts and $\mathcal{C} \subseteq \{(p_i, p_j) | p_i, p_j \in \mathcal{P}\}$ is the corresponding set of relations between parts. A simple tube assembly consisting of four parts is shown in Figure 2.1a.

A common assembly representation is as "graph of connections" [53] or liaison diagram [29] $G_{\mathcal{A}} = (N_{\mathcal{A}}, E_{\mathcal{A}})$, where the relations correspond to contacts between parts. In this representation, the set of parts $N_{\mathcal{A}}$ is used as the node set, and the relations $E_{\mathcal{A}}$ are used as edges between them. Figure 2.1b depicts the liaison diagram for the tube assembly.

Parts might carry additional information such as their type (e. g., screw, angle bracket, etc.) or their pose relative to the assembly [96, 97, 10]. In particular, if the assembly representation is used in combination with learning-based ASP algorithms, the parts may also include geometric information represented using various 3D shape descriptors [127].

Relations might also encode how two parts are attached to each other [52, 30, 96, 97] (e.g., glued together, clip attachment, screwed, etc.). In cases where the parts are only in contact but not attached, the contact type might be encoded instead (e.g., planar surface on planar surface, cylinder in hole, etc.). Some representations also encode the location of the relation either directly [52] or by splitting the parts into surfaces and then establishing relations between these surfaces instead of between parts [97]. Besides being in close proximity, relations between parts can also indicate other spatial relationships, such as colinearity and coplanarity [127], or parallelism and orthogonality [10]. This can lead to edges connecting parts that are far apart in the physical assembly, potentially resulting in a fully connected graph.

Another common representation of an assembly is via matrices. All graphs can also be represented as an adjacency matrix, which is a square matrix where each column and row corresponds to a node. The entries in the matrix represent the presence or absence of edges between pairs of nodes, with a value of 1 indicating the presence of an edge and a value of 0 indicating its absence. The adjacency matrix for the graph $G_{\mathcal{A}}$ is called the liaison matrix [74]. As stated above, relations might encode several additional pieces of information, which can also be stored in matrix form by mapping the categorical data onto integers. For example, different contact types can be represented by distinct integer values, which are then stored in a separate matrix with the same dimensions as the liaison matrix [30].

2.1.2 Representation of Assembly Sequences

An assembly sequence is a sequence of assembly states, where one common way to encode such a state is via set partitions [53] of the set of all parts. That is, as the assembly process progresses, the set partition becomes coarser until all parts are in one set in the final assembled state. An alternative is to use boolean strings of fixed sizes, where each position corresponds to a liaison [29].

Independent of the chosen assembly state representation, a common representation of multiple assembly sequences is as a DAG [53]. Each node corresponds to an assembly state in one of the representations discussed above, and each edge to the assembly steps

required to progress from one state to the next. Another common, but more compact representation is as an AND/OR graph [51]. Here, each node corresponds to a subassembly, which is represented by the set of its constituent parts. Each hyperarc represents an AND relationship, meaning that all the subassemblies pointed to by a hyperarc must be present to form the parent assembly. If a node has multiple outgoing hyperarcs, this represents an OR relationship, meaning that any one of these hyperarcs can be followed to create the parent assembly. Examples of an assembly graph represented as DAG and as an AND/OR graph are provided by Figure 2.1c and 2.1d, respectively.

2.1.3 Conditions for Feasible Assembly Sequences

For an assembly sequence to be feasible, all assembly steps must satisfy certain conditions. These are either precedence relations or establishment conditions [53], where the former one dictates what connections must be established before which other connection. In contrast, the latter one informs what connections can be established starting from a given assembly state.

It has been shown in [53] that given a complete and correct set of either precedence relations or establishment conditions, one can derive a complete set of feasible assembly sequences and vice versa. The former set of conditions is known as implicit [53] or constraint-based [121] representation of assembly sequences. In contrast, any assembly representation discussed in the previous sections from which one can directly derive an assembly sequence is known as an explicit [53] or enumerative [121] representation. While an explicit representation is more straightforward to interpret, an implicit representation is often more compact.

Furthermore, precedence relations and establishment conditions can be understood as high-level constraints that encompass a variety of more specific feasibility constraints [59]. Fundamental ones are geometric feasibility and stability of subassemblies. These ensure that parts can be added without collision and that intermediate assemblies remain stable with respect to gravity during the assembly process. Depending on the level of detail in the planning process, additional constraints may include ensuring that a manipulator can assemble a part without colliding with itself or other parts, as well as the availability of specific tools, such as screwing end effectors.

2.1.4 Properties of Assembly Sequences

An assembly sequence can have several properties [120, 121, 88]:

- **Number of hands:** This property indicates the number of manipulators (hands) required to add subassemblies from different directions during a single assembly step. An additional hand is assumed to hold all parts that remain fixed during this step.
- **Sequential:** If an assembly sequence is sequential (or two-handed), during each assembly step, only a single part (or subassembly) is added to the current assembly state.

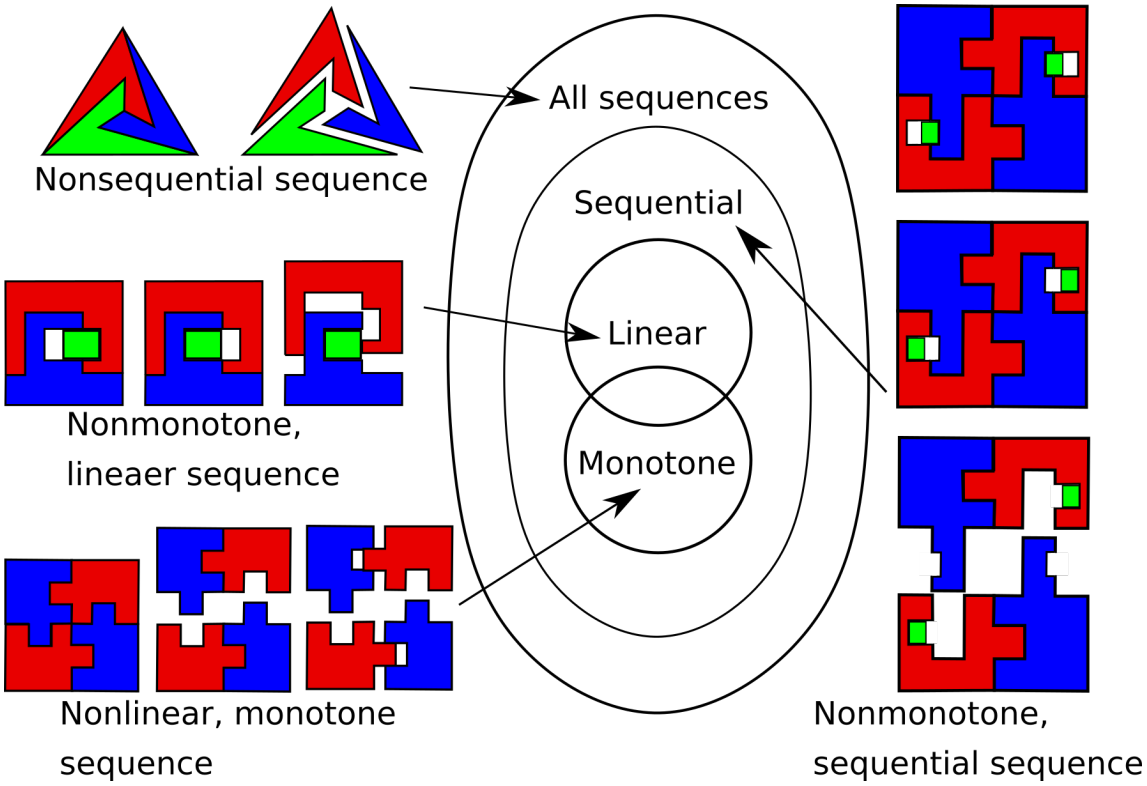


Figure 2.2: Assembly sequences with various properties (adapted from Wolter [120]. © 1989, IEEE).

- **Linear:** In a linear assembly sequence, there are no subassemblies. That is, only a single part is added to the current assembly state during each assembly step. All linear assembly sequences are also sequential.

- **Monotone:** An assembly sequence is monotone if parts already added to the assembly (i.e., all parts present in the current assembly state) do not need to be moved to add new parts. This excludes, for example, deformable parts or parts that require intermediate positions during the assembly process.

- **Contact-coherent:** An assembly sequence is contact-coherent if each part added (except the first) makes contact with at least one previously assembled part.

Descriptive examples of assembly sequences with these properties are shown in Figure 2.2. These properties provide important information about the complexity of planning and executing an assembly sequence. For example, non-sequential assembly sequences require the planner to consider adding several parts from various directions during the same assembly step. Additionally, the system that executes the sequence must control a sufficient number of manipulators to handle the simultaneous addition of multiple parts or subassemblies.

If not indicated otherwise, for the remainder of this work, we discuss assemblies for which linear, monotone, and contact-coherent sequences are sufficient. An example is the tube assembly shown in Figure 2.1a.

2.2 Generating Feasible Assembly Sequences

As stated in Section 2.1.3, feasible assembly sequences can directly be derived from a set of precedence relations. Several works directly learn such relations, for example, by posing questions about the assembly to humans [15, 29], or by learning them from human demonstration [115, 35, 36, 34, 99].

However, a disadvantage of these approaches is that they require demonstrations from a human expert. Therefore, a significant body of research has focused on extracting precedence relations from CAD models [30, 81, 118, 43, 119, 45, 106, 103, 26, 77, 38, 32, 33, 107]. These methods utilize the connection between precedence relations and feasibility checks to automatically determine feasible assembly sequences. Specifically, through simulating the assembly process, they can, after each assembly step, test if the step itself, as well as the resulting assembly state, are feasible, i.e, the part could be added without collision and the resulting state is stable.

These methods are limited by the fact that they have to start the planning process from scratch for each new assembly. Therefore, various approaches focused on reusing knowledge from previous planning attempts either through CBR [72], where similar past cases (here, feasible assembly sequences for subassemblies) are retrieved and adapted to solve new problems [93, 21, 92, 105, 78, 97, 127], or by directly learning to predict what assembly actions are feasible [125, 41, 83, 10].

In the following, we will review these approaches in more detail, starting with methods that learn from humans, either through asking questions or from demonstration. We will then discuss various approaches, which utilize CBR. Afterwards, we examine methods for extracting precedence relations through simulation of the assembly process. Following this, we explore techniques that directly learn feasible assembly actions. In the final section, we provide a summary and discussion of all reviewed methods.

2.2.1 Direct Learning of Precedence Relations from Human Experts

Bourjault [15] pioneered a method to obtain precedence relations by questioning human experts. DeFazio and Whitney [29] refined this technique by reducing the number of required questions and making them more intuitive.

Another strategy is LfD, a vast research area [14, 73], which we will focus on in relation to ASP. In general, an LfD for ASP system involves several steps. First, an expert demonstrates one or more successful assembly sequences, typically by physically manipulating the parts or through a graphical interface. The system then learns a representation of the demonstrated assembly sequences, which is subsequently generalized to handle similar assembly tasks, while accounting for variations in part geometries, numbers of parts, and other factors. Finally, the generalized assembly plan is executed on a robotic system.

A hierarchical approach to Multimodal Assembly Skill Decoding (MASD) was introduced by Wang et al. [115]. After a demonstration was captured using a stereo camera system, the demonstrated skills were identified in two stages: first, individual actions were recognized using multimodal information (gesture, trajectory, and action-object effects) with a top-level Support Vector Machine (SVM) classifier [22]. Second, dynamic

programming was used to segment the continuous demonstration into multiple assembly skills, assuming a library of assembly skills is available. Using this library, the system then matched the identified skills to corresponding robot code templates. These were then filled with specific parameters, also extracted from the demonstration, to create executable robot programs. MASD was evaluated on two assemblies: a flashlight and a light switch. When compared against several other action recognition methods, it demonstrated high accuracy and real-time performance.

A novel technique for identifying actions from bimanual human demonstrations was proposed by Dreher et al. [35]. Specifically, given a demonstration of a task recorded with an RGB-D camera, the authors first extracted object and hand poses. Using the depth data, they then estimated 3D bounding boxes for the detected objects and hands. From these bounding boxes, they determined various spatial relations between objects, such as "above" or "inside". These relations, along with object identities, were used to construct a scene graph representing the current state. In addition, they captured temporal information by concatenating scene graphs from ten consecutive frames. The resulting graph structure served as input to a Graph Neural Network (GNN) [12], which learned to predict the action being performed by each hand independently. The approach was evaluated on the KIT Bimanual Actions Dataset [1], which contains data from five kitchen tasks, such as preparing breakfast cereals and cooking, as well as four workshop tasks, including disassembling a hard drive and hammering nails. During evaluation, when considering only its top prediction, the model correctly identified the action being performed approximately 63% of the time. This was further improved to about 86% when considering the top three predictions.

Building upon this work, Dreher et al. [36] later developed a more comprehensive multisensory, robot-supported platform designed to capture human demonstrations during complex disassembly processes. This enhanced system was used to record 15 participants performing 10 disassembly trials each on different variants of an electric motor. The platform captured RGB-D video streams from both a stationary camera and a mobile camera mounted on a robot arm, as well as 3D human pose data, object detections, and eye tracking data.

Both Wang et al. [115] and Dreher et al. [35] focused on identifying an action sequence from a single demonstration. Wang et al. [115] also then identified assembly skills consisting of multiple actions, which were, however, predefined. In contrast, Dreher and Asfour [34] proposed a model-driven approach that automatically creates temporal task models from multiple bimanual human demonstrations. These models represent precedence relations at two levels: between tightly coupled sets of actions (thus automatically identifying skills (subtasks in [34])) and between these sets of skills. The models are implemented as fully connected directed graphs, where nodes represent actions and edges track the frequencies of observed temporal relations between actions across all demonstrations. The system was evaluated on two datasets: one based on the electric motor disassembly scenario discussed in Dreher et al. [36], and the previously mentioned KIT Bimanual Actions Dataset [35]. They measured the precision and recall of identified temporal constraints. The results showed high recall from the beginning and increasing precision as more demonstrations were added to the model. Furthermore, to demonstrate its real-world applicability, a temporal task model describing a clean-up task was successfully executed on the humanoid robot ARMAR-6.

[1] https://bimanual-actions.humanoids.kit.edu/

A three-level LfD system that integrates learning at the sensorimotor, semantic, and planning levels was proposed by Savarimuthu et al. [99]. At the lowest level, motor skills were learned via Dynamic Movement Primitives (DMPs) [56]. At a semantic level, the sequence of changes in object relations during an action was extracted and represented as Semantic Event Chains (SECs) [6]. At the highest level, the system used probabilistic planning operators to learn the preconditions, effects, and success likelihoods of actions. Assembly tasks were demonstrated using a teleoperation setup where a human performed actions with magnetically tracked objects while a robot mimicked the movements in real-time. In this way, it was possible not only to record the trajectories and object poses but also to measure the forces and torques the robot exerted. The authors demonstrated the capabilities of their approach by teaching a robot to assemble the Cranfield Assembly Benchmark, a standard assembly task involving ten parts, including pegs, plates, and separators. We will discuss several such benchmarks at the end of this chapter.

2.2.2 Reusing Precedence Relations via Case-Based Reasoning

One approach to knowledge transfer is CBR [72]. The basic idea behind this approach is to maintain a library of cases. Each one is represented in a structured format that captures the essential features of a previously solved problem as well as its solution. If a new problem needs to be solved, a CBR system will use a retrieval mechanism to identify relevant cases from the library. Typically, a similarity measure is defined and used to find the most similar matches between the features of the new problem and those stored in the library. Next, an adaptation mechanism will modify the solutions of retrieved cases to fit the specifics of the new problem. The solution is then validated. Finally, suppose a valid solution was found in this way. In that case, a case retention process decides whether and how to incorporate the new problem-solution pair into the library, which is crucial for the system's ability to learn from experience.

An early application of CBR to ASP was discussed by Pu and Reschberger [93]. Their system, CAB-Assembler, utilizes a small library of primitive cases representing individual assembly steps rather than complete assembly sequences. These cases are described by feature vectors that capture all involved parts along with their associated spatial and geometric constraints. When retrieving a case, the authors proposed not only matching the features but also taking into account how often the case's solution could either be used to solve a new problem or fail to do so. The retrieved assembly steps are then combined, and each one is validated by asking a human expert during the training phase. This allows the system to detect and learn from failures, updating the success and failure counts for each case. The authors demonstrated that their approach could efficiently solve ASP problems, particularly for enclosure-type assemblies, and that training the system on more complex problems enabled it to solve simpler ones without failures.

Another early work is presented by Chakrabarty and Wolter [21], who state that all assemblies can be decomposed into a "hierarchy of standard structures". They introduced a case library that contains solutions for four of such structures, each involving two parts: "against", "threaded", "fits", and "blocks". Their planner takes as input a user-provided hierarchical decomposition of the assembly with labeled standard structures and then uses the library to retrieve solutions for these structures. It then iteratively merges solutions for substructures, backtracking if necessary, until a solution for the complete assembly is

found. The authors proved that their planner is both correct and complete, and demonstrated its effectiveness on an example assembly, showing improved performance compared to a system [120] that does not reuse solutions.

Pu and Purvis [92] extended the approach of Pu and Reschberger [93] by formalizing the ASP problem as constraint satisfaction problem (CSP). Specifically, they used a repair-based constraint satisfaction solver [85] to find an assembly sequence that satisfies the feasibility constraints and, thus, precedence relations. Instead of starting from scratch, this solver can begin with an initial, potentially incorrect, solution and then repair it. This initial solution is derived from similar cases in the case library. Importantly, this combination of CBR and CSP techniques lifts the need for human validation during the adaptation process, which was a limitation of [93].

Swaminathan and Barber [105] introduced an Assembly Planner using Experience (APE). To encode a case, they extracted loops from the mating direction graph, which is an assembly's contact graph with directed edges. That is, each edge has an attribute indicating the direction along which the parent node can be mated with the corresponding child node. APE represents each case in its database as a loop family, which encompasses all mating direction strings that can be generated from a reference string through four transformations: "reorient", "swivel", "reverse", and "cyclic". When planning a new assembly, APE first matches encountered loops to these families. It then uses the transformation parameters to adapt the stored plans to the specific orientation and traversal of the new assembly's loops. Finally, the plans are merged such that they satisfy every part's precedence relationships. The resulting solutions are represented as Assembly Precedence Graphs, which are DAGs where nodes represent parts and edges indicate precedence relations and mating directions. This is in contrast to methods discussed by Pu and Reschberger [93] and Pu and Purvis [92], where only a single assembly sequence was planned.

Lee et al. [78] proposed an approach for generating assembly sequences for product families. Unlike typical CBR approaches, their primary goal was not to accelerate the planning process. Instead, they aimed to generate sequences that closely resemble the case assembly sequence. The authors assumed that this case sequence represents the ideal assembly sequence for a group of products from the same family. Their method begins with a case-based preplanning stage, utilizing a modified version of OAKplan [100] to generate assembly sequence candidates based on past cases. The authors then created a relation matrix based on geometric information extracted from the product's CAD model. Each generated assembly sequence candidate was then rated using this matrix. Finally, the best-scoring candidate was adapted, if necessary, to solve the current assembly problem. The authors evaluated their approach using a toy airplane assembly. The experimental results confirmed that their approach can produce suitable assembly sequences for multiple assemblies within the same product family, even with limited reference cases.

Rodríguez et al. [97] introduced the Knowledge Transfer - Robotic Assembly Sequence Planner (KT-RASP) algorithm, which utilizes a graph-based representation of assembly topology. In this representation, nodes correspond to either parts or their surfaces, while edges represent contacts between them. While each assembly corresponds to a specific topology, a single topology can represent multiple assemblies. Each topology defines a unique mapping between parts and a parameter vector that captures the relative distances between surfaces. The authors focused their study on assemblies composed of aluminum profiles connected via angle brackets. They built a case library of 5000 assemblies, each containing two or three profiles. For every assembly, they computed precedence rela-

tions (referred to as rules in the paper) using an automatic rule deduction algorithm [96]. New topologies were added to the database if they enclosed rules not covered by existing topologies. For known topologies, new assemblies were stored along with their specific parameters and associated rules. The assemblies were then grouped into configuration classes based on their rule sets. For a new assembly, the KT-RASP algorithm matches it to known topologies in its database, potentially identifying multiple instances of known topologies within a larger assembly. For each identified instance, the system extracts the parameter vector. For each one, a neural network classifier then predicts the corresponding configuration class and set of precedence rules. Finally, the rules from all identified instances are combined. The authors evaluated their approach on 20 test assemblies, with the largest consisting of five profiles and five angle brackets. Using KT-RASP significantly reduced planning times compared to a traditional method discussed by Rodríguez et al. [96], particularly for more complex assemblies, while maintaining solution correctness in 39 out of 40 cases across two test scenarios.

Another graph-based approach to matching new assemblies to those stored in a case library was discussed by Zhou et al. [127]. The authors introduced a graph representation where each node stores the shape of a part, encoded via a rotation-invariant 3D shape descriptor [62, 9], as well as its order in the reference assembly sequence. Edges in this graph represent assembly relations, established between parts that are either coplanar or collinear. For a new assembly, each part is matched to parts in the reference assembly using a two-step process. First, parts are compared based on the similarity of their shape descriptors. For parts that exceed a shape similarity threshold, a second comparison is made based on the number of assembly relations (i.e., the number of edges the corresponding node has) within their respective assemblies. This second criterion is also used for parts that could not be matched based on shape. If neither criterion is met, the system requests assistance from a human expert. After completing the matching process, the assembly order from the matched reference parts is used to generate a candidate sequence for the new assembly. This candidate sequence is then refined using the contact graph of the new assembly. During refinement, the order of identical parts may be swapped when necessary to maintain feasibility with respect to gravity constraints. The authors evaluated their method on three classes of wooden chair assemblies. Each one included one reference assembly and two variants with differences in component shapes and numbers. The method successfully generated feasible assembly sequences for most variants. However, the system required human input to handle entirely new components (e.g., a backrest added to a stool design).

2.2.3 Inferring Precedence Relations via Feasibility Checks

In this section, we provide an overview of methods that search for feasible assembly sequences by simulating the assembly process using CAD models of the assembly. These approaches test each potential assembly step against various feasibility constraints, with two fundamental constraints being geometric feasibility and stability. Geometric feasibility determines whether a part or subassembly can be added to the current assembly state without causing a collision. At the same time, stability ensures that the current state remains stable with respect to gravity.

Particularly for testing geometric feasibility, a large body of prior work exists [58]. In the following, we will first briefly discuss the advantages and disadvantages of simulating the

assembly process in either a forward or backward direction. Subsequently, we will review approaches that derive blocking relationships between parts, followed by an examination of methods that utilize sampling-based path planners.

Forward ASP vs. Backward ASP

When searching for feasible assembly sequences in simulation, there are two directions in which an assembly can be constructed:

- **Forward assembly:** This method starts with a single part or subassembly and successively adds parts until the assembly is complete. It is intuitive and reflects the real-world assembly process. However, it can encounter dead ends from which it must backtrack.
- **Backward assembly (also known as AbD):** This approach starts from the complete assembly and iteratively removes parts until all have been removed. The disassembly sequence is then inverted to obtain an assembly sequence. This assumption holds that assembly and disassembly are the inverse of each other, which is valid as long as no permanent joining methods, such as gluing or welding, are used. Because it starts from the fully assembled state and works backwards, this approach automatically eliminates many infeasible sequences.

Approaches That Derive Blocking Relationships Between Parts

We will now review several approaches that precompute the blocking relationships between parts. These methods determine which parts obstruct or block the movement of others along various directions. The resulting relationships are then stored in specialized data structures that enable efficient lookup of removable parts. Specifically, if any other part does not block a part, it can be removed from the assembly.

Dini and Santochi [30] simulated the movement of parts along the x-, y-, and z-axes of a Cartesian coordinate system and recorded collisions (i.e., blocking relationships) in two types of matrices: the contact matrices recorded if a part's movement is immediately blocked by another part, indicating a contact between these two parts. The interference matrices, on the other hand, captured collisions that occur during the simulated movement. Furthermore, the authors used a third matrix type, connection matrices, to represent the types of connections between components, such as threaded connections or O-rings, as integers. For each direction and matrix type, a separate matrix is used, resulting in a total of nine matrices. The authors presented an algorithm that utilizes these matrices to systematically evaluate the feasibility of potential subassemblies. The result is a hierarchical structure of subassemblies, with individual parts forming the lowest layer. Explicit assembly sequences can then be recovered from this structure via a depth-first search (DFS).

Lu et al. [81] introduced the Local Constraint Digraph (LCD), a directed graph that records blocking relationships between parts for a specific assembly direction $\vec{d}$. In an LCD, each node represents a part, and a directed edge from node n_i to n_j indicates that part p_i is blocked by part p_j when moved along direction $\vec{d}$. It follows that part p_i must be assembled before part p_j along this direction. Loops in the LCD indicate subassemblies. The authors' method recursively computes LCDs for each subassembly, resulting in a hierarchy of subassemblies similar to the approach of Dini and Santochi [30].

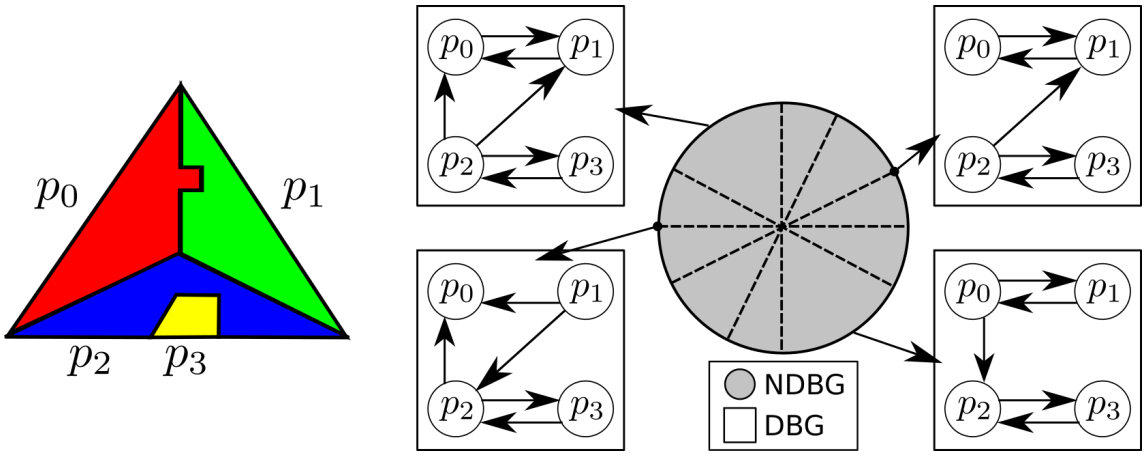


Figure 2.3: Non-directional blocking graph (NDBG) with directional blocking graph (DBG) for 2D infinitesimal translations (adapted from Wilson and Latombe [118]. © 1994 Published by Elsevier B.V. CC BY-NC-ND 4.0[2]).

Wilson and Latombe [118] observed that LCDs, which they called directional blocking graphs (DBGs), remain constant for sets of motions. They introduced non-directional blocking graphs (NDBGs) to represent these sets. The authors considered three cases: infinitesimal translations, infinitesimal translations and infinitesimal rotations, and infinite translations. In all cases, these sets are computed by examining the interactions between each pair of parts in the assembly. Specifically, potential collisions between two parts are determined through two methods: for infinitesimal translations and/or rotations, by analyzing the contacts between parts; and for infinite translations, by computing the Minkowski difference

$$\mathcal{Q}_{p_0} \ominus \mathcal{Q}_{p_1} = \{q_0 - q_1 \mid q_0 \in \mathcal{Q}_{p_0}, q_1 \in \mathcal{Q}_{p_1}\}, \tag{2.1}$$

where $\mathcal{Q}_{p_0}$ and $\mathcal{Q}_{p_1}$ represent the sets of all the points on the surface of the two parts. This difference identifies all translations that would cause a collision between the parts. These collision-causing motions are then represented geometrically. For translations in 2D, the directions are projected onto the unit circle and, for translations in 3D, the surface of the unit sphere. When rotations are included, the directions are projected onto a 5D unit hypersphere. The intersections of these projections from all part pairs create a partition of these spheres into regions, each corresponding to a set of motions with a constant DBG. Figure 2.3 illustrates an example of a NDBG for infinitesimal translations, along with some of its corresponding DBGs.

An algorithm to efficiently compute an NDBG for infinitesimal motions was described by Guibas et al. [43]. Instead of computing all regions, it focuses on computing directions that lie inside "maximally covered" regions (cells in [43]). They have the property that moving infinitesimally outside the region into neighboring regions would result in a reduction of the number of blocking relationships. Therefore, it is sufficient to only analyze the DBG associated with these regions.

To generalize the NDBG approach, Wilson et al. [119] introduced the interference diagram. It removes the requirement that motions are either infinitesimal or infinite. First, a common coordinate system is chosen for all parts in the assembly. Then, for every pair

[2] https://creativecommons.org/licenses/by-nc-nd/4.0/

of parts, the configuration space (C-space) obstacle is computed, i.e., the set of all configurations where one part intersects with the other part. As previously discussed, if only translations are considered, this set is computed using the Minkowski difference defined by Equation 2.1. However, instead of projecting these sets onto a sphere, they are superimposed in the previously chosen common coordinate system. This is the key difference in the construction of a NDBG. The boundaries of the superimposed sets divide the space into regions, where each one is labeled with the C-space obstacles that include it. Using these labels, a DBG can be computed for each region. Because a common coordinate system was used, any point in the diagram corresponds to the placement of an arbitrary part or subassembly. To test if a placement in a region is feasible, one must check the corresponding DBG to ensure that there are no outgoing edges from the node or nodes encoding the part or subassembly. It follows that a removal path is only feasible if all regions through which this path goes have DBGs that allow the movement of the part or subassembly being considered. To find a removal path for a subassembly, one starts from the initial configuration and explores neighboring regions. The DBG of the starting region is updated as new regions are added to the path. A path is considered feasible if its combined DBG (the union of DBGs of all regions along the path) is not strongly connected, meaning there exists a subassembly that can move along the entire path without being blocked.

Halperin et al. [45] combined the work of Wilson and Latombe [118], Guibas et al. [43] and Wilson et al. [119] into a general ASP framework. Specifically, the concept of motion space (M-space) was introduced, which is a parametric representation of all motions that separate a subassembly or a part from the remaining assembly. Each point in the motion space corresponds to a path, starting from a fixed initial position, along which a subassembly or a part can potentially be moved. Therefore, its dimension corresponds to the minimum number of parameters required to uniquely define such a path, given this fixed starting point. For example, if no restrictions are placed on the motions, the dimension of the motion space is infinite. If the motions are one-step translations of predefined length, then the M-space is two-dimensional. Motion spaces are tightly related to the C-space. As explained above, one can compute regions in the M-space (M-regions), which, for pairs of parts, describe along which motions they would collide.

An approach similar to the motion space one was discussed by Thomas et al. [106]. They computed the C-space obstacles for pairs of parts using the Minkowski difference described by Equation 2.1 and then projected them onto two half-spheres using a stereographic projection. However, instead of constructing DBGs, they preserved the information on motion length during the projection. In fact, they stored it in two $2\frac{1}{2}$D distance maps, where each entry corresponds to a direction. Then, using this information, they build an AND/OR graph, discussed in Section 2.1.2, by systematically testing if a subassembly or a part could be separated.

Approaches Using Sampling-Based Path Planning

In this section, we review methods that utilize sampling-based path planners to determine removal paths for individual parts or subassemblies. The main idea behind these approaches is to explore the C-space of the assembly by randomly sampling collision-free configurations and connecting them to construct feasible paths. Two well-known algorithms are probabilistic roadmap (PRM) [61] and Rapidly-exploring Random Tree (RRT) [75]. PRM first precomputes a roadmap by randomly sampling collision-free con-

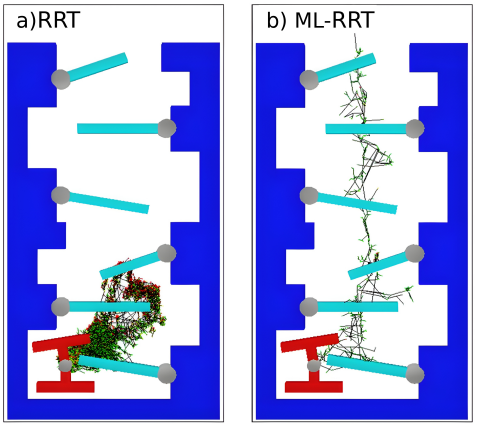


Figure 2.4: RRT vs ML-RRT (adapted from Cortés et al. [26]. © 2008, IEEE).

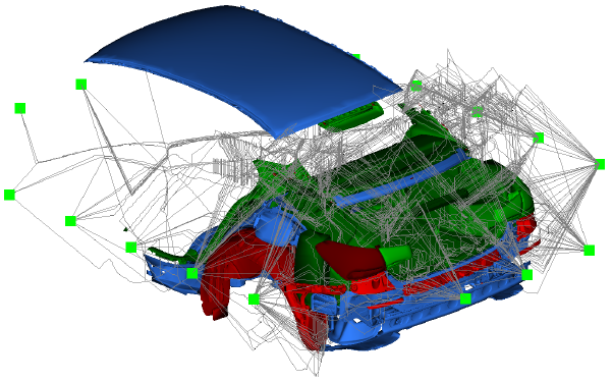


Figure 2.5: Example of a GVD-based roadmap (reproduced from Dorn et al. [31]. © 2020, IEEE).

figurations from the C-space and connecting these configurations to their nearest neighbors. Then, during the actual planning phase, the precomputed roadmap is used to connect the start configuration to the end configuration. In contrast, RRT finds a path between start and goal configurations by incrementally constructing a random tree. It iterates through the following steps: First, it samples a random collision-free configuration from the C-space. Then, it identifies the nearest neighbor to this sampled configuration among the existing nodes in the tree (initially, this is only the start configuration). Finally, it attempts to move from the nearest neighbor towards the sampled configuration for a predefined distance. If successful, a new node is created and connected to the nearest neighbor node, expanding the tree. This continues until a path connecting the start and goal configurations is found.

Sundaram et al. [103] presented a method for ASP based on PRM [61]. In this context, the C-space includes all possible poses of each part in the assembly. For example, an assembly with two parts that require both translation and rotation in 3D results in a 12-dimensional C-space. The PRM is then used to plan a path from the assembled product to a fully disassembled state. The authors highlighted that disassembly sequences often involve tightly packed configurations, which introduces the narrow passage problem [55]. This problem complicates the search for collision-free paths using random sampling. To mitigate this, the authors leveraged the geometric properties of the parts, biasing the sampling process by moving parts along directions aligned with the normals of their faces. The effectiveness of this approach was demonstrated on several puzzle-like assemblies consisting of two to twelve parts.

Cortés et al. [26] extended the RRT algorithm by developing the Manhattan-like RRT (ML-RRT). This new algorithm was specifically designed to plan disassembly paths for two articulated parts composed of multiple connected elements. They observed that the standard RRT algorithm often produces inefficient paths with unnecessary movements when dealing with articulated assemblies due to the complexity of the C-space. To improve efficiency, they introduced a distinction between active and passive configuration parameters. Active parameters refer to the positions and orientations of the parts directly controlled by the planner. In contrast, passive parameters represent the joint variables of the articulated elements, which are adjusted only when they hinder the motion of the

active parts. This distinction allowed the algorithm to focus on the essential movements required for disassembly and minimize redundant motions. Their experiments demonstrated that ML-RRT significantly outperformed the basic RRT algorithm, especially as the number of articulated elements increased. A comparison of both algorithms for a 2D assembly is shown in Figure 2.4.

Le et al. [77] observed that the core idea behind ML-RRT is more general than its original application to two articulated parts. They suggested the Iterative-ML-RRT (I-ML-RRT) algorithm. As its name suggests, it iteratively applies the ML-RRT algorithm to the individual parts of an assembly, where active parameters correspond to the part (or parts if subassemblies are considered) that should be disassembled. In contrast, the position and orientation of the remaining parts are treated as passive parameters. This enables I-ML-RRT to solve monotonic as well as non-monotonic disassembly problems as we discussed in Section 2.1.4. Furthermore, because the algorithm can make progress even if it tries to remove a part that is blocked by other parts, it can efficiently explore various disassembly sequences without getting stuck. When the authors compared I-ML-RRT with the method proposed by Sundaram et al. [103], it outperformed the latter by two orders of magnitude in computational efficiency.

Ebinger et al. [38] presented a general framework for disassembly sequence planning. It relies on two data structures: a disassembly graph and a roadmap. The disassembly graph provides a high-level representation of the disassembly process, where nodes store the assembly states and edges indicate feasible transitions between these states. The roadmap, on the other hand, stores low-level information about the removal paths, with nodes representing specific configurations of parts and edges representing feasible paths between these configurations. One key idea is that the framework uses multiple separation tests with varying computational complexity. In detail, the authors employed two tests: the first attempts part removal along predefined directions derived from the geometry of the parts, similar to the method proposed by Sundaram et al. [103], while the second uses RRT [75] to search for more complex removal paths. Additionally, the framework includes a novel method for identifying subassemblies based on the first separation test. That is, through analyzing collision information obtained during simple linear translation attempts, the method groups parts into subassemblies. The authors discussed two methods to expand the disassembly graph: a preemptive DFS for efficiently finding feasible sequences and a full breadth-first search (BFS) to find optimal sequences. Comparative results showed significant performance improvements over the I-ML-RRT [77] method, particularly in solving more complex assemblies.

Similarly to the approach presented by Sundaram et al. [103], the Expansive Voronoi Tree (EVT) method proposed by Dorn et al. [32] centers on precomputing a roadmap of potential disassembly paths for use during ASP. An example of such a roadmap is depicted in Figure 2.5. Their method employs a General Voronoi Diagram (GVD) to create this initial roadmap. An ordinary Voronoi diagram partitions a plane into Voronoi cells, each containing all points closest to a specific seed point. It follows that the cell boundaries are equidistant from neighboring seeds, thus providing paths with maximum clearance. The GVD generalizes this concept and can use arbitrary 3D meshes as seeds. Dorn et al. [31] presented an efficient algorithm for computing the GVD. Using this GVD-based roadmap, the authors first searched for the shortest path. However, as this path may not always be feasible for moving a part, they employed the Expansive Space Tree (EST) algorithm [54] for local refinement. This combination of global path planning via the

GVD and local optimization with EST enables the EVT to efficiently find disassembly paths through open spaces and narrow passages. It was evaluated on a subset comprising 18 parts of a real-world car assembly dataset and outperformed several other planners in both path length and reliable running time.

Dorn et al. [33] expanded on their previous work [32] by presenting an ASP framework for complex real-world assemblies. Their approach shares similarities with Ebinger et al. [38] in using two distinct strategies to evaluate part separability. However, they do so to handle flexible parts. In detail, they split the planning into two phases: a NEAR phase, which employs the Iterative Mesh Modification Planner [48] to locally plan with flexible fastening elements, and a FAR phase that utilizes the EVT for efficient planning of the remaining path. The authors introduced two techniques to reduce the number of failed feasibility tests. First, they implemented a path existence check using the GVD-based roadmap, which quickly assesses if a potential disassembly path exists. Second, they employed a bookkeeping heuristic that tracks parts that could not be removed in previous attempts. These parts are only tested again after the removal of one or more parts against which they previously collided. The framework was evaluated on a larger subset (comprising 661 parts) of the car assembly also used by Dorn et al. [32]. It outperformed a RRT-based approach in both path-finding capability and efficiency. Also, the heuristics proved highly effective, reducing the overall calculation time by approximately 67%.

A physics-based ASP framework was introduced by Tian et al. [107]. Specifically, the authors customized a physics simulator that utilizes a penalty-based contact model to handle collisions between parts. It requires an efficient way to compute the penetration distance and speed. The original simulator had the limitation that in each collision pair, at least one object had to be a primitive shape with a known distance function. To support complex geometries, the authors implemented signed distance fields, which they computed for each collision shape to serve as distance functions. For disassembly planning, the framework employs a BFS-guided algorithm that iteratively applies one of six unit forces or unit torques to the part for a predefined timestep and observes its movement in the physics simulation. For multi-part assemblies, the algorithm iterates over all parts, trying to disassemble each one, starting with a shallow search depth that is incrementally increased if no parts can be removed. Like the approach by Ebinger et al. [38], this prioritizes simpler disassembly paths and can quickly identify and remove less constrained parts first. The authors evaluated their method on thousands of industrial assemblies. Their results showed that it, particularly with progressive deepening BFS, achieved superior performance in both success rate and computational efficiency compared to baseline approaches, including the work by Ebinger et al. [38].

2.2.4 Improving Precedence Relation Inference via Knowledge Transfer

In the previous section, we explored methods that simulate the (dis)assembly process while testing the feasibility of each step. In contrast, in this section, we review approaches that accelerate precedence relation inference by leveraging knowledge from previous ASP attempts to directly predict the feasibility of assembly steps.

We begin by introducing the Markov Decision Process (MDP) [104], a mathematical framework for modeling sequential decision-making. One common approach to solving

MDPs is RL, and we will discuss deep Q-learning (DQL) [86] in detail. Next, we describe how to formalize the ASP process as an MDP. Finally, we review several methods that leverage knowledge transfer to enhance the efficiency of ASP.

Markov Decision Process

The MDP [104] is a mathematical framework for modeling decision processes. It is described by the tuple (S, A, P, R). In this formulation, S represents the set of possible states, while A corresponds to the set of available actions. The transition probability $P(s_t, a_t, s_{t+1})$ indicates the likelihood of moving from state s_t to state s_{t+1} when action a_t is taken. Each transition is associated with an immediate reward $R(s_t, a_t, s_{t+1})$.

The actual decision-making is modeled via a policy $\pi : S \to A$, which is a function that maps states to actions. The expected cumulative reward for a policy is then computed as

$$\mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t R(s_t, a_t, s_{t+1})\right], \tag{2.2}$$

where $a_t = \pi(s_t)$ and γ a discount factor. The solution to an MDP is then given by the optimal policy π^* that maximizes this expected reward.

Deep Q-learning

Q-learning [104] is one approach to finding a solution for an MDP, i.e., an optimal policy π^*. Given a policy π, the Q-function $Q^\pi(s, a)$ estimates the expected reward for taking an action a in state s and thereafter following the actions as determined by π. It is also known as the state-action function. The optimal Q-function $Q^*(s, a)$ uses the optimal policy π^* to determine actions and satisfies the Bellman optimality equation:

$$Q^*(s_t, a_t) = \mathbb{E}\left[R(s_t, a_t, s_{t+1}) + \gamma \max_{a_{t+1}} Q^*(s_{t+1}, a_{t+1})\right]. \tag{2.3}$$

The main idea behind Q-learning is to use this equation to define an iterative update:

$$Q_{i+1}(s_t, a_t) = \mathbb{E}\left[R(s_t, a_t, s_{t+1}) + \gamma \max_{a_{t+1}} Q_i(s_{t+1}, a_{t+1})\right], \tag{2.4}$$

where Q_i converges against Q^* as i approaches infinity. After the Q-function has converged, the optimal policy π^* can be derived from it by selecting the action that gives the highest Q-value for each state:

$$\pi^*(s) = \arg\max_{a \in A} Q^*(s, a). \tag{2.5}$$

Traditionally, the Q-function is often implemented as a table. However, this becomes impractical for large state and action spaces due to the exponential growth of possible state-action pairs. Instead, DQL, as introduced by Mnih et al. [86], uses a neural network to approximate the Q-function, parameterized by θ, such that $Q(s, a; \theta) \approx Q^*(s, a)$.

To train the neural network, DQL first collects past transitions (s_t, a_t, r_t, s_{t+1}) and stores them in an experience replay buffer. It then samples minibatches from this buffer. This is

done to break the temporal correlations between samples, which improves the stability of the training. The minibatches are then used to minimize the following loss:

$$L(\theta_i) = \mathbb{E}_{(s_t,a_t,r_t,s_{t+1})\sim\mathcal{D}}\left[(y_i - Q(s_t, a_t; \theta_i))^2\right], \tag{2.6}$$

where $\mathcal{D}$ represents the distribution of transitions sampled from the replay buffer, and the target value y_t is computed as:

$$y_i = R(s_t, a_t, s_{t+1}) + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1}; \theta_{i-1}). \tag{2.7}$$

Here, θ_{i-1} represents the parameters of a separate target network, which are a delayed copy of the Q-network's parameters θ that are periodically updated. Besides the experience replay buffer, using a target network is another technique to stabilize the training.

While we have used the θ parameter here to emphasize that the Q-function is implemented via a neural network and to simplify our explanation of the training process, for simplicity, we will drop this notation for the remainder of this work.

Formalizing ASP as an MDP

We can formalize ASP as a finite MDP. The set S comprises all assembly states $s \subseteq \mathcal{P}$, where each state contains the parts currently present in the assembly. An action is defined as mounting a part to the assembly, with $a_t = i$ representing adding part $p_i \in \mathcal{A}$ at timestep t. In the case of ASP by AbD, the actions correspond to removing parts. Furthermore, we define the transition function P and reward function R as follows:

$$P(s_t, a_t, s_{t+1}) = \begin{cases} 1, & \text{if } s_{t+1} = s_t \oplus a_t \text{ and the action } a_t \text{ is feasible,} \\ 1, & \text{if } s_{t+1} = s_t \text{ and the action } a_t \text{ is infeasible,} \\ 0, & \text{otherwise,} \end{cases} \tag{2.8}$$

where $s_t \oplus a_t$ denotes the state resulting from applying action a_t to state s_t:

1. **Forward ASP (assembling parts):** $s_{t+1} = s_t \cup p_i$
2. **Backward ASP (disassembling parts):** $s_{t+1} = s_t \setminus p_i$.

Action a_t is feasible if part p_i can be added to (or removed from) state s_t without collision, and the resulting state s_{t+1} remains stable under gravity. That is, if a feasible action is performed, we move to the next state s_{t+1} resulting from that action. Otherwise, if the action is infeasible, we remain in the same state $s_{t+1} = s_t$.

We now define the reward function R such that feasible actions are rewarded, while infeasible actions are penalized:

$$R(s_t, a_t, s_{t+1}) = \begin{cases} 1 & \text{if the action is feasible} \\ -1 & \text{if the action is infeasible} \end{cases} \tag{2.9}$$

Deep Learning Approaches to Predict Precedence Relation

We will review several approaches that utilize knowledge learned during past planning attempts to predict which assembly actions are feasible. One way to do so is to formalize the ASP process as an MDP and then learn a Q-function via DQL to guide the planning

process, as was done by Zhao et al. [125] and Ghasemipour et al. [41]. Another learning paradigm, supervised learning, was used by Ma et al. [83] and Atad et al. [10]. It tries to learn a direct mapping between actions and their feasibility. That is, unlike RL, supervised learning does not consider the future effects of these actions, which is equivalent to setting the discount factor γ to 0 in Equation 2.7.

Zhao et al. [125] proposed a deep RL approach for ASP for workpieces (ASPW-DRL). The system takes two inputs: an image of the current assembly state and an image of the potential next part to assemble. Using a deep Q-network implemented with a convolutional neural network (CNN), it predicts whether the part should be attached. The assembly process is simulated in Gazebo [71], a 3D robotics simulator. A positive reward is provided only if the entire assembly process is completed. Otherwise, a negative reward is given. To address the challenges of sparse rewards, the authors employ an experience replay buffer, as well as curriculum learning, to gradually increase task difficulty. The approach was initially tested on a seven-part car model, achieving 100% accuracy. To evaluate generalizability, the authors tested the model on three assemblies: a toy model with five parts, a humanoid model with eight parts, and a house model with nine parts. However, they retrained the model on each new assembly before conducting the test. For each retrained model, they conducted 100 tests with randomly generated feasible initial states, and the system achieved a 100% success rate for each of them.

Ghasemipour et al. [41] introduced a physics-based environment for multi-part assembly tasks. Their goal was to assemble diverse structures from a fixed set of components. Specifically, the authors create a 3D simulated environment containing 16 cuboid blocks of six different types, each with magnetic connection points. A pair of free-floating grippers directly manipulate these blocks to assemble structures according to a predesigned blueprint. The authors represented the environment state as a fully connected graph, where most nodes correspond to blocks and store their height, as well as whether each gripper was holding them during the last timestep. The edges carry the majority of the information, including the change in position and orientation required to align the correct magnets, the difference between their current positions and orientations, and those specified by the blueprint, as well as whether the blocks should be or are currently connected. Two global nodes store the observations for each gripper, i.e., their orientation, positional and rotational velocities, as well as the block they were holding in the last timestep. This graph representation was then processed with a Graph Attention Network (GAT) [111], which is a type of GNN [12] that incorporates attention mechanisms to weigh the importance of neighboring nodes and edges when making predictions. The GAT produces three types of outputs: per-block motion actions, per-block key vectors, and per-gripper block query vectors. The per-block motion actions are 6D vectors representing how each block should be moved if selected. To match grippers to blocks, the authors compute the dot-product attention between the per-gripper query vectors and the per-block key vectors. They evaluated their system by training on 141 blueprints and testing on 24 unseen ones, which contained between 2 and 16 blocks. While their approach could solve all training blueprints and successfully generalize to the unseen test blueprints, it required a substantial amount of training data.

Ma et al. [83] introduce a graph-transformer framework for ASP for LEGO models. The authors collected 100 user-created LEGO animal models uploaded to LEGO Studio, then manually cleaned them by adjusting colliding parts. They describe a graph-based assembly representation, where nodes represent individual LEGO bricks and edges represent

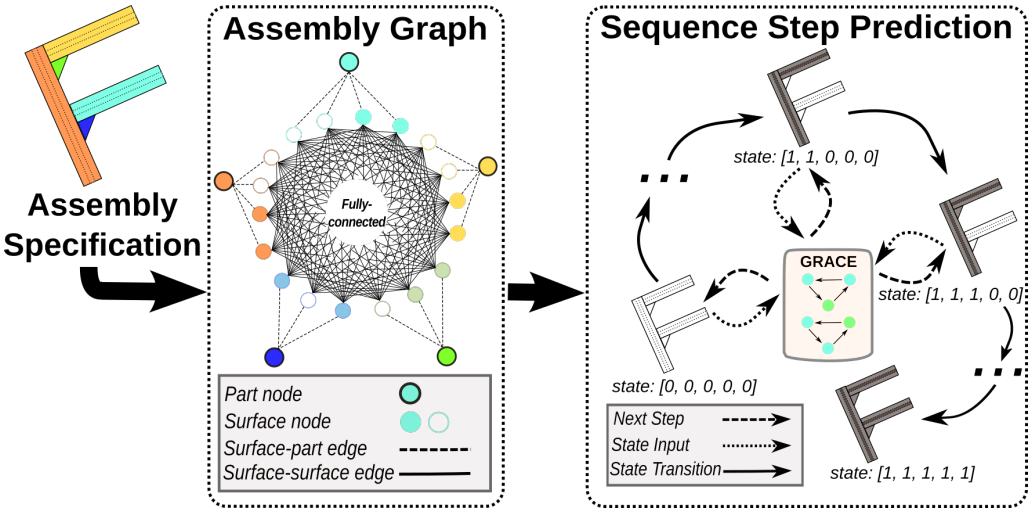


Figure 2.6: Overview of the ASP approach for aluminum profile assemblies proposed by Atad et al. [10]. Each profile's surface is represented as a node in a fully connected graph. This graph serves as input to their Graph Assembly Processing Network (GRACE), which predicts the assembly sequence step-by-step (adapted from Atad et al. [10]. © 2023, IEEE).

relationships between bricks. Each node has two types of features: a 24-dimensional, one-hot encoded vector representing the brick type and a normalized 3D position of the brick's center. They use two types of edges: link edges, which represent knob-cavity connections between bricks, and block edges, which represent spatial blocking relationships along the x-, y-, and z-axes. The authors proposed a graph-transformer architecture to predict assembly sequences from these graphs. In detail, a graph transformer [37] is an extension of the traditional transformer model [110] to handle graph-structured data. A key feature of transformer models is their use of attention, which enables the computations of a node embedding to be more influenced by the values of specific neighbors rather than all neighbors. The authors computed separate attention values for different types of edges to handle the heterogeneous nature of their LEGO assembly graph representation. These values are then aggregated to produce the final node embeddings. Finally, these embeddings are used as input for a standard decoder, a component of the transformer architecture, which generates sequences by predicting each element based on all the previously predicted ones. To evaluate the system, the authors measured the correlation [63] between the predicted and ground-truth assembly sequences. Their results showed a moderate correlation for smaller graphs (using the smallest 60% of the dataset) with a degrading performance for larger, more complex graphs.

Graph assembly processing networks (GRACE) were introduced by Atad et al. [10]. Similarly to Rodríguez et al. [97], their work focused on planning assembly sequences for aluminum profiles. Furthermore, the assembly graph representation in GRACE is quite similar to the topology-based approach discussed by Rodríguez et al. [97]. Both methods use nodes to represent parts and surfaces, and both capture relative distances between surfaces. However, while the topology representation only connects surfaces in contact, GRACE employs a fully connected graph structure with edges between all surface nodes. In Rodríguez et al. [97], relative distance information is encoded in parameter vectors associated with each specific instance of a topology. In contrast, here the authors store this data directly in the edge features of their graph. They then use this graph representa-

tion as input to a GAT [111] which predicts feasible assembly sequences in a step-by-step manner. An overview of this process is depicted in Figure 2.6. To evaluate GRACE, they created a dataset of profile assemblies ranging from three to seven parts. In intra-sized experiments, the model was trained and tested on assemblies of the same size, while inter-sized experiments assessed generalization across different assembly sizes. For intra-sized assemblies, GRACE achieved perfect step prediction performance and near-perfect sequence prediction for smaller assemblies, with slight decreases for larger ones. In inter-sized tests, GRACE demonstrated strong generalization capabilities, particularly when applying knowledge from larger to smaller assemblies. However, it showed a performance drop when generalizing to larger, potentially more complex structures.

2.2.5 Discussion

In this section, we provided an overview of several approaches to generate feasible assembly sequences. We first discussed methods to obtain such sequences from human experts. In fact, early works in the field by Bourjault [15] and DeFazio and Whitney [29] directly asked experts to create assembly sequences. Other works, e.g., by Wang et al. [115] or Dreher and Asfour [34], leveraged LfD techniques to extract sequences from human demonstrations.

Another approach to creating assembly sequences that we discussed next is CBR. When presented with a new assembly, CBR first decomposes it into cases, i.e., subassemblies. It then searches a library for similar subassemblies. By adapting and combining solutions to these similar cases, CBR generates an assembly sequence for the new assembly. While it can expedite the planning process, this approach presents two main challenges. First, it requires maintaining a case library, which can be a resource-intensive task. Second, determining the optimal size for cases might be difficult: if cases are too small, they may not capture enough context about neighboring parts to be useful. Conversely, if cases are too large, the number of variations that must be stored in the library grows exponentially, which might make retrieval and adaptation more complex and time-consuming.

We then examined two techniques that simulate the assembly process using CAD models. Both of them test that various feasibility constraints, particularly geometric feasibility and stability, are satisfied after each assembly step, thereby determining which are physically possible and in what order they must occur.

The first technique involved precomputing blocking relationships between parts and storing them in specific data structures. Examples of such structures are NDBG introduced by Wilson and Latombe [118] or $2\frac{1}{2}$D distance maps proposed by Thomas et al. [106]. They allow for the quick determination of which parts can be (dis)assembled in the current assembly state, thus enabling efficient ASP. While the precomputation of blocking relationships enables efficient ASP, it becomes computationally expensive for arbitrary 6D motions. In fact, most previously discussed methods assume either infinite or infinitesimal motions, which limits their ability to handle complex motions.

The second technique employs sampling-based motion planners, such as PRM [61] or RRT [75], for finding feasible (dis)assembly paths. While more flexible than the first technique, these algorithms may require a significant amount of time to find complex paths. Consequently, researchers have long been working on accelerating this process through various methods. Earlier approaches we discussed included moving parts only

Table 2.1: Overview of learning-based methods for predicting feasible assembly actions reviewed in Section 2.2.4. † The model was retrained for each new assembly during evaluation. ‡ Intra-sized experiments: The model was trained and tested on assemblies of the same size. Inter-sized experiments: The model was trained on assemblies of a certain size and then tested on assemblies with different sizes.

	ASP direction	**Direct sequence prediction**	**Dataset**			**Uses robotic manipulator**	**Part representation**
			Type of parts	**Training**	**Evaluation**		
Zhao et al. [125]	Forward	✓	Cubes, cylinders, prisms	Toy model (5 parts)	† Car (7 parts), humanoid (8 parts), house (9 parts)	✗	Image
Ghasemipouret al. [41]	Forward	✓	6 types of blocks with magnetic connectors	141 assemblies (2 - 16 parts)	24 assemblies (2 - 16 parts)	Two free-floating grippers	Block height
Ma et al. [83]	Forward	✓	LEGO bricks	75 LEGO assemblies (3 - 44 parts)	25 LEGO assemblies (3 - 44 parts)	✗	One-hot vector
Atad et al. [10]	Forward	✓	Aluminum profiles, angle brackets	17513 assemblies (3 - 7 parts)	‡ Cross-validation: intra-sized and inter-sized experiments	Two robotic manipulators	One-hot vector
This thesis	Backward (AbD)	✗	Mechanical components: aluminum profiles, screws, wheels, …	5 real-world assemblies (29 - 68 parts)	Leave-one-out cross-validation	✗	3D shape descriptor

after collisions occurred with other parts, as proposed by Cortés et al. [26] and Le et al. [77], and utilizing removal checks with increasing complexity as presented by Ebinger et al. [38]. More recent contributions include the precomputation of potential paths explored by Dorn et al. [32][33] as well as using simulated normal forces to guide the path search as suggested by Tian et al. [108].

However, these algorithms start from scratch for each new assembly, which can be inefficient. An alternative approach leverages knowledge from previously solved assembly problems to guide the planning process for new assemblies. We reviewed several methods that implement this strategy, as summarized in Table 2.1.

All of these methods focus on directly predicting assembly sequences without incorporating any recovery strategies for incorrect predictions. While this is the most efficient method for generating assembly sequences, it has its limitations. It works well for shorter sequences, as shown by Atad et al. [10], but requires substantial training data, as demonstrated by Ghasemipour et al. [41], to handle more complex assemblies.

The lack of a recovery strategy means this approach struggles with completely unseen parts and requires that precedence relations for all parts can be learned from the training set. If this condition is not met, the approach fails, as Ma et al. [83] discovered when attempting to generalize from smaller to larger LEGO assemblies with more complex structures.

This limitation is further reflected by the fact that three of the four discussed works assumed known part types. Specifically, Ma et al. [83] and Atad et al. [10] represented

different part types using a one-hot vector, while Ghasemipour et al. [41] assumed that all parts were blocks and then utilized their height. This raises questions about the generalizability of these approaches to more diverse and complex real-world assembly tasks. Additionally, all these approaches predicted sequences in a forward direction. Therefore, they always needed to consider the complete assembly (both assembled and not yet assembled parts) during each prediction step. This is in contrast to the backward (AbD) direction, where removed parts can be ignored during all future predictions.

2.3 Optimizing Assembly Sequences

Manufacturers are often interested not only in feasible assembly sequences but also in sequences optimized for various objectives. An overview of these objectives is provided by Jones et al. [60]. Common optimization goals include minimizing the number of tool changes, reducing assembly direction changes, and decreasing overall assembly time.

A basic approach to finding optimal sequences involves exhaustive search [11, 38, 51, 7]. However, given the combinatorial explosion, there are $N!$ potential sequences for an assembly with N parts, this approach is only useful for small assemblies. More advanced approaches use soft computing techniques such as GA [76, 109, 84, 122, 68] or RL [67, 89] to efficiently find near-optimal solutions in a reasonable time. Nevertheless, these methods have to start each planning attempt from scratch.

Similar to feasible ASP, an alternative approach is to reuse knowledge from previous planning attempts to guide the sequence optimization for a new assembly [116, 47, 44, 39].

In this section, we first provide an overview of different approaches to optimizing assembly sequences, ranging from basic exhaustive search methods to more sophisticated algorithms designed to efficiently navigate complex search spaces. In the final section, we explore approaches that reuse previous planning knowledge, offering potential improvements in efficiency and effectiveness.

2.3.1 Exhaustive Search

A straightforward approach to finding optimal assembly sequences is to exhaustively search all feasible options as discussed by Bahubalendruni and Biswal [11]. When these sequences are represented as a DAG, standard graph-search algorithms such as BFS can be applied, as demonstrated by Ebinger et al. [38]. For more efficient exploration of the search space, heuristic-based algorithms like A* are often employed. An extension of A*, known as AO*, is specifically designed for AND/OR graphs [90], and was, for example, used by Homem de Mello and Sanderson [51]. However, they still first construct the whole AND/OR assembly graph.

Andre and Thomas [7] presented an innovative anytime ASP algorithm that can be stopped and return a feasible sequence at any time while continuously improving the solution as long as computation time is available. Their approach used precomputed $2\frac{1}{2}$D distance maps [106] to incrementally construct an AND/OR graph while simultaneously searching

for the optimal sequence. They optimized the weighted sum of two objectives: minimizing the number of tool changes required during the assembly process and reducing the number of part reorientations needed to complete the assembly. The authors successfully demonstrated the functionality of their algorithm on 4 assemblies: a cross and a rectangle consisting of aluminum profiles (13 parts and 24 parts, respectively), a gearbox (8 parts), and a drilling machine (26 parts).

2.3.2 Approximating Optimal Sequences via Soft Computing Algorithms

In this section, we provide an overview of methods that approximate optimal assembly sequences using soft computing algorithms. These approaches address the combinatorial explosion by utilizing heuristics to guide the search towards potentially rewarding areas in the search space. Their application to ASP has been extensively studied [28].

We will first review various approaches utilizing GA. Afterwards, we discuss methods that employ RL.

Genetic Algorithm

GA solve optimization problems through an evolutionary-inspired iterative process. Starting with a population of potential solutions encoded as "chromosomes" the algorithm repeatedly applies selection, mutation, and crossover steps. Selection chooses solutions stochastically based on their fitness, which normally corresponds to the value of the objective function. This favors fitter solutions while still maintaining a diverse solution population. The mutation step introduces small random changes to some selected solutions, while the crossover step recombines parts of chromosomes from pairs of selected solutions. Each iteration creates a new generation, with the population evolving over time. The process continues until a stopping criterion is met, such as reaching a maximum number of generations or achieving a desired fitness level.

Lazzerini et al. [76] employed a GA for assembly planning, assuming an assembly system that can only attach parts along the vertical axis. Their approach encoded three key properties in each chromosome: the part assembly order, their corresponding vertical insertion directions (upward or downward), and the gripper types used to grasp them (e.g., vacuum or two-finger grippers). They randomly created the initial population. Their fitness function considered sequence feasibility, orientation changes, gripper changes, and grouping of similar assembly operations, such as screwing or pressing. Feasibility is evaluated using interference, contact, and connection matrices [98]. Their mutation step used three operators: The first exchanges one or two randomly selected genes (where a gene is an element of a chromosome) in the part order, mirroring these changes in the insertion direction and gripper type variables. The second randomly replaces genes in a selected portion of the insertion direction variable. The third randomly changes a single gripper type to another suitable option. The authors introduced a modified partially matched crossover [42] that allows matching and exchanging portions of to-be-mated chromosomes located at different positions. The method was tested on assemblies with up to 16 components, consistently converging to feasible sequences. In 63 of 100 trials for an 8-component example, it found the optimal solution.

Given an AND/OR graph representing all feasible assembly sequences, Valle et al. [109] presented a GA that minimizes the total assembly time. It considers assembly duration, time for tool changes, and subassembly transportation times. For creating the initial population, they used a heuristic to estimate a lower bound for this time. In contrast, to compute the actual fitness of a chromosome, they created an assembly schedule and then simulated it. The GA employs two types of mutation operators: "reordering tasks", which performs local search by moving tasks within a sequence, and "replanning mutation", which explores different assembly sequences by replacing subtrees in the AND/OR graph. For crossover, they introduced a "replanning crossover" operator that combines tasks from two parent chromosomes while maintaining task positions when possible. The algorithm was evaluated on a hypothetical product consisting of 30 parts. Results showed that using all three operators yielded the best performance, consistently improving solutions over time, with the heuristic-based initialization further enhancing the algorithm's effectiveness.

Marian et al. [84] proposed a GA that relies on a guided search to generate the initial population of feasible sequences as well as to implement its mutation and crossover operators. Its chromosomes represent the order of assembly operations, which the authors defined over the Entities Meaningful for Assembly Sequence (EMAS) instead of parts. That is, EMAS can represent not only individual parts but also subassemblies, assembly operations, or special processes, such as adding fluids. This allows the GA to handle a broader range of assembly scenarios, including nonsequential, nonlinear, and nonmonotone assembly processes. We discussed these properties in Section 2.1.4. The guided search algorithm utilizes a liaison graph, where vertices represent EMAS and edges represent liaisons or relationships between EMAS, to determine candidates for each assembly step. It then checks, for each candidate, if assembling it satisfies additional constraints such as geometric feasibility, accessibility, and additional assembly process requirements. The fitness function uses a penalty-based approach, assigning penalties for suboptimal assembly steps (e.g., rotating heavy components). Their crossover operator combines features from both parents while maintaining feasibility, using guided search to fill in missing genes. Instead of traditional mutation, they implement a "pseudo-mutation" that randomly replaces chromosomes with new feasible ones generated by a guided search. The authors tested their approach on a hydraulic linear motor with 25 parts. The GA consistently converged to optimal or near-optimal solutions within reasonable computation times.

Wu et al. [122] proposed a discrete differential genetic algorithm (DDG) for ASP. Unlike standard GA, DDG replaces the traditional sampling step with a discrete difference operation. The DDG algorithm begins by randomly generating an initial population, where each chromosome represents an assembly sequence encoded as an integer vector. The main loop then iterates over these chromosomes, applying three key operations. First, in the "variation step", three chromosomes (P_1, P_2, P_3) are randomly selected. The algorithm calculates the positional differences between similar genes in P_1 and P_2, then uses these differences to reorder the genes in P_3, creating a new chromosome V. Second, a "crossover operation" is performed with a certain probability between V and the current chromosome, using a partial cross-recombination method. It randomly selects a crossover point, exchanges subsequences between the parent chromosomes, and then resolves any conflicts by removing duplicate genes. Third, a "mutation operation" may be applied, randomly swapping two genes in V. After these operations, the fitness of V is evaluated and compared with the current chromosome. The fitness function is computed as a weighted sum of three objectives, considering only feasible assembly sequences: the number of

assembly direction changes, the number of assembly tool changes, and the stability of the assembly process. To determine feasibility and calculate the objectives, the algorithm uses interference and connection matrices [30]. The chromosome with the higher fitness is retained for the next generation. The authors compared their algorithm against a standard GA using a vise assembly consisting of 11 parts. Results showed that DDG consistently outperformed GA in terms of convergence speed and solution quality, with DDG able to find the optimal solution more frequently and with smaller population sizes.

A proper multi-objective optimization approach to ASP based on the Non-dominated Sorting Genetic Algorithm II (NSGA-II) [27] was proposed by Kiyokawa et al. [68]. Unlike previous works that optimized weighted sums of objectives, the authors optimized the Pareto front for two objectives: optimizing insertion conditions and minimizing the constraint state transition difficulty (CSTD). Optimizing insertion conditions prioritizes sequences following natural insertion orders, e.g., adding a part with a hole before assembling a peg. Minimizing CSTD focuses on adding parts that are less constrained in the current assembly state. The Pareto front represents the set of non-dominated solutions, where each solution is better than all others in at least one objective. To check sequence feasibility and compute objectives, the authors used three matrices extracted from 3D CAD models: The "interference-free matrix" shows whether one part can move without colliding with another. The "insertion matrix" captures insertion relationships, such as plug-and-socket connections. The "degree of constraint matrix" measures movement restrictions between parts, considering translational and rotational constraints. For rigid parts, these matrices are computed directly from geometric data. For deformable parts, such as rubber bands, the authors simulated potential deformation before extracting the matrices. The approach was evaluated on eight different assemblies with up to 33 parts, of which several were deformable, e. g., rubber bands, belts, and chains. For all eight assemblies, it could consistently generate feasible assembly sequences.

Deep Reinforcement Learning

As described in Section 2.2.4, when using rewards for sequence completion or the execution of feasible actions, RL can be utilized to train policies that select feasible actions at each assembly state. However, if the reward function is defined to reflect specific optimization objectives, such as part accessibility or the number of tool changes, deep RL can be used to plan assembly sequences that are optimized with respect to these objectives. The policy is then trained and evaluated on the same assembly.

Kitz and Thomas [67] used DQL to plan assembly sequences that are optimized with respect to the accessibility of the individual parts. To compute this objective and to speed up the planning process, the authors used the $2\frac{1}{2}$D distance maps introduced by Thomas et al. [106]. They represent the assembly state as a boolean vector where a 1 indicates that the part is still present in the assembly and a 0 means it has been removed. The action space consists of selecting the next part to remove. To implement a learnable Q-function, the authors used an Feed-Forward Neural Network (FNN) with N input and output nodes (where N is the number of parts), which allows the direct mapping of the assembly state to Q-values for each possible removal action. The search process follows an ϵ-greedy strategy, with the ϵ value being gradually decreased during training to balance exploration and exploitation. Each episode continues until the assembly is completely disassembled. Significantly, the Q-function is updated after each episode, i.e., the learning process is directly integrated into the search. The authors evaluated their approach on

three different assembly tasks: a small Soma cube (a 3D puzzle consisting of 7 parts), an item-profile construction (16 parts), and a large Soma cube (56 parts). Their results showed that the approach could quickly generate feasible assembly sequences and then progressively improve the accessibility of individual parts.

Neves and Neto [89] applied four different RL algorithms to finding optimal assembly sequences for a toy airplane. They were tabular Q-Learning [104], DQL [86], Advantage Actor-Critic (A2C) [87], and Rainbow [50]. The airplane consists of 11 distinct components and two types of fasteners, which were grouped into eight assembly tasks. Similarly to Kitz and Thomas [67], the state representation used a boolean vector for completed tasks. Additionally, the currently selected tool was encoded as an integer. The action space comprised the eight assembly tasks. The authors discussed a scenario in which some assembly tasks are performed by a human expert and others by a robot. Therefore, the reward function combined total assembly time minimization with user preferences to avoid ergonomically challenging sequences. The algorithm was evaluated in both deterministic and stochastic assembly time settings. Their results showed that A2C, Rainbow, and tabular Q-Learning achieved near-optimal performance. Surprisingly, DQL performed poorly compared to the other algorithms, including tabular Q-Learning. The authors suggested that its poor performance might be due to infrequent updates of the target network, which leads to outdated target values and slower learning. Furthermore, the experience replay might have hindered learning by delaying the use of recent experiences and continuously reusing transitions from initial poor policies. However, the paper lacks a detailed discussion of hyperparameters such as the update frequency of the target network or the size of the experience replay buffer.

2.3.3 Knowledge Transfer

While in the last section, we reviewed works that trained and evaluated policies on the same assembly, here we discuss approaches that transfer knowledge from previous planning attempts to new assemblies. Specifically, through utilizing pretrained policies, these methods can accelerate the learning process and potentially achieve better optimization outcomes.

An early approach to transferring knowledge from one assembly to another was proposed by Watanabe and Inada [116]. Their goal was to find an assembly sequence that minimizes the time required by two robotic manipulators to assemble wooden block models. The authors used a traditional tabular Q-learning [104], where the state was encoded as a binary number. Actions were represented as pairs of numbers indicating which blocks to disassemble with the left and right robot hands, respectively. The Q-values were updated based on rewards that considered sequence completion, individual block removals, and penalties for infeasible operations. To encourage faster sequences, a higher reward was provided if the current disassembly time was shorter than the best time found so far. Additionally, simultaneous disassembly operations using both robot hands were rewarded more than single-hand operations. To transfer knowledge between assemblies, the authors experimented with two methods: a regression analysis, where Q-values from simpler wooden assemblies served as the dependent variable and 20 features describing block properties and assembly states as independent variables. For the other method, they trained an FNN that took the same 20 features as input to predict Q-values as output. Both methods, once

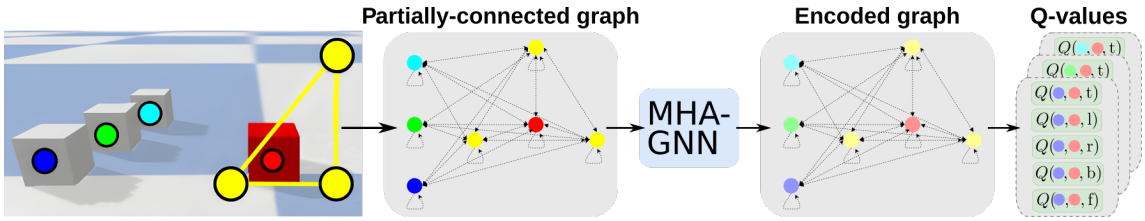


Figure 2.7: Illustration of Funk et al. [39]'s approach, where cubes (blue, green, and cyan) should be assembled to fill a 3D target shape (yellow). The assembly is represented as a graph, which is processed by a multi-head attention graph neural network (MHA-GNN) to compute an embedding. A trained Q-function uses this embedding to predict the optimal placement of each cube relative to already placed cubes (red) (adapted from Funk et al. [39]. © 2022, IEEE).

trained on simpler models, were used to initialize Q-tables for more complex new assemblies. After training the system on five simpler assemblies, it was evaluated on two unseen assemblies with seven and ten blocks, respectively. The authors compared their transfer learning approaches with a baseline Q-learning method that does not utilize transfer. Results showed that both methods significantly outperformed the baseline, with the neural network approach demonstrating the best performance.

Hayashi et al. [47] presented an approach for minimizing the amount of temporary supports required to build spatial trusses using an AbD strategy. They represented the truss as a graph, with the state defined by a tuple (C, v), where C is the connectivity matrix encoding member-node relationships and $\vec{v}$ is a matrix containing binary flags for support conditions and local stability for each member. The authors then used a graph embedding to transform this variable-sized state representation into fixed-size feature vectors. These are used as input to an FNN that estimates Q-values for each possible action, i.e., which member to remove next in the disassembly process. The parameters of both the graph embedding and the Q-value estimation network are trained via DQL. The authors trained the agent on various randomly generated truss configurations and validated it on unseen structures. Their approach significantly outperformed GA and Covariance Matrix Adaptation Evolution Strategy (CMA-ES) [46], an evolutionary algorithm for continuous optimization, in terms of solution quality and computational efficiency. In fact, it found better assembly sequences over a thousand times faster. The method was validated on three unseen structures: a dome truss with 25 nodes and 56 members, a flat roof truss with 28 nodes and 80 members, and a large latticed shell with 220 nodes and 797 members.

Guo et al. [44] introduced the Connector-Linked Model (CLM), a graph-based assembly representation based on precedence graphs rather than liaison graphs. In CLM, nodes represent connectors (e.g., screws, bolts) and edges indicate assembly precedence. In detail, node attributes encode the connector ID and type, the precedence position, as well as the required assembly tool and assembly direction. The edge attributes capture the number of changes in connector type, assembly tool, and assembly direction between connected nodes. The assembly state was represented by a matrix derived from the CLM and encoded using an edge graph attention model. The action space involved selecting the next connector to assemble, along with its tool and assembly direction. A decoder then calculated a probability distribution over viable connector combinations, based on the state encoding and the encoding of the previously assembled connectors. The entire

Table 2.2: Overview of learning-based methods for predicting optimal assembly actions reviewed in Section 2.3.3. † A learned Q-function was combined with an MCTS during decision time to provide it with a one-step lookahead.

	ASP direction	Direct sequence prediction	Dataset			Uses robotic manipulator	Optimization objectives	Pareto optimization
			Type of parts	Training	Evaluation			
Watanabe and Inad [116]	Forward	✓	Blocks	5 models (3 - 7 parts)	Model A (7 parts) Model B (10 parts)	Two robotic manipulators	Assembly duration; Sequence completion	✗
Hayashi et al. [47]	Backward (AbD)	✓	Trusses (members are connected to nodes)	5000 random variation of flat trusses or dome trusses	Dome truss (25 nodes; 56 members), flat roof truss (28 nodes; 80 members), latticed shell (220 nodes; 797 members)	✗	Number of temporary supports	✗
Guo et al. [44]	Forward	✓	Connectors (e.g., screw, bolt, . . .)	One million synthetic assemblies (20 - 50 connectors)	One thousand synthetic assemblies (20 - 50 connectors) Two real-world assemblies (19 and 24 connectors)	✗	Number of changes in: connector type, assembly tool, assembly direction	✗
Funk et al. [39]	Forward	†✗	Blocks of various shapes	Random target shape to fill with 20 - 34 blocks		Robotic manipulator	Fill-in of the target structure; stability	✗
This thesis	Backward (AbD)	✗	Aluminum profiles	14 small and 7 large assemblies (21 and 30 parts)	Leave-one-out cross-validation	Robotic manipulator	Removal path length	✗
		✗	Blocks of various shapes	8 small and 4 large Soma cubes (38 and 58 parts)	Two-fold cross-validation	✗	Part accessibility; number of direction changes	✓

model is trained using the REINFORCE algorithm to minimize changes in connector type, assembly tool, and assembly direction across the sequence. The system was evaluated on synthetic datasets with varying numbers of connectors (20-50), using one million instances for training and one thousand for testing at each size. Additionally, it was tested on two real-world reducer assembly cases: one with 24 connectors and another with 19 connectors. The results showed that the approach achieved lower assembly costs and faster decision-making times compared to other algorithms, namely pointer networks [112], GA, and particle swarm optimization, particularly for larger problem instances.

Funk et al. [39] described an approach to robotic architectural construction. Specifically, they addressed the challenge of how a robot can assemble a set of wooden cubes to best fill out a given 3D target shape. The authors represented the workspace as a graph, where each node corresponds to either a target point defining the desired shape, an already assembled wooden cube, or a cube that still needs to be added to the structure. Edges are established from unplaced cubes to placed cubes as well as to target points. To process this graph representation, the authors used a multi-head attention graph neural network (MHA-GNN). The attention mechanism helps the GNN to focus on important relationships between nodes. Multi-head attention extends this concept by computing multiple attention operations in parallel, each potentially focusing on different aspects of the graph structure. The authors utilized the graph representation to train a Q-function that assigns Q-values to assembling an unplaced cube relative to already placed cubes. This process is illustrated in Figure 2.7. They then explored various methods to integrate this learned Q-function with MCTS. These methods include using MCTS only during testing to enhance the learned policy, incorporating MCTS during both training and testing, and investigating different strategies for balancing exploration and exploitation within the MCTS framework. Through extensive experiments, they demonstrated that combining the learned Q-function with MCTS at decision time provides the best performance and generalization across diverse assembly tasks.

2.3.4 Discussion

We have reviewed a range of approaches for planning assembly sequences optimized for various objectives. Initially, we considered methods that generate all feasible sequences and then evaluate them to identify the optimal one. While straightforward, these approaches are limited by the exponential growth of the search space and are, therefore, only practical for small assemblies.

This limitation has driven researchers to explore the application of various soft computing algorithms to optimal ASP. We gave an overview of two such algorithms: GA and RL. Instead of exhaustively exploring the entire solution space, they employ heuristics and learning mechanisms to efficiently guide the search.

Parallel to our review of feasible assembly sequence generation in the previous section, we next explored approaches that reuse knowledge learned during previous planning attempts to efficiently optimize assembly sequences for new assemblies. An overview is given by Table 2.2.

Similar to the approaches that reuse knowledge to accelerate feasible assembly sequence planning, as shown in Table 2.1, all of these optimization methods, except for Funk et al. [39], directly predict the sequence using the learned Q-function. However, in contrast

to planning feasible assembly sequences, when planning optimal sequences, selecting a suboptimal action can lead to either an infeasible sequence or one that is suboptimal with respect to the optimization objectives. To avoid the former case, two of the reviewed approaches, Hayashi et al. [47] and Guo et al. [44], restricted the action space to only allow the selection of feasible actions. This ensures that a feasible sequence is always generated, although it might not be optimal.

Conversely, the strategies discussed by Watanabe and Inada [116] and Funk et al. [39] could produce infeasible sequences. Notably, while Funk et al. [39] did not use the Q-function to predict the sequence directly, but instead combined it with an MCTS, they analyzed a scenario where a robotic manipulator directly executed each planned assembly step. Therefore, the authors focused on rapid prediction and limited the search depth of the MCTS to a single assembly step. Moreover, because of this scenario, once an assembly action was taken, the planner could not backtrack.

Another important point is that, among all the reviewed works, only Kiyokawa et al. [68] employed proper multi-objective optimization. Specifically, while most authors combined multiple objectives into a single scalar value using weighted sums, [68] utilized NSGA-II [27] to optimize the Pareto front, which represents the set of non-dominated solutions. That is, each solution in this set is better in at least one objective compared to all others. Providing a set of solutions, rather than a single one, offers valuable insight to human operators. Moreover, it allows them to select the solution that best balances different priorities based on the specific manufacturing context. This is particularly useful in scenarios where priorities might change or when it is challenging to determine appropriate weights for balancing multiple objectives.

2.4 Assembly Datasets

To accelerate the ASP process via deep learning, training datasets are required. While research on 3D shape retrieval [57] has led to the creation of numerous datasets containing large numbers of common objects, only a few of those contain mechanical parts [13, 64, 69] and even fewer mechanical assemblies [82, 117].

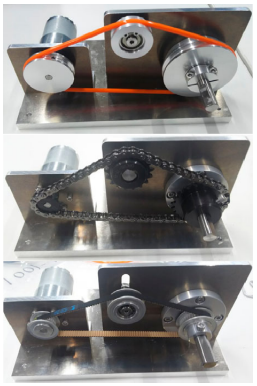
Additionally, robotic assembly benchmarks [25, 17, 65, 79, 24] have been developed to standardize the evaluation of assembly planning and execution algorithms. These benchmarks typically provide specific assembly tasks with well-defined metrics. However, they often offer only a single benchmark or a few variations, which limits their usefulness for training deep learning models.

In this section, we will provide an overview of existing resources for assembly planning. First, we will discuss datasets focused on individual mechanical parts. Following this, we explore datasets that consist of complete assemblies. Finally, we discuss various robotic assembly benchmarks.

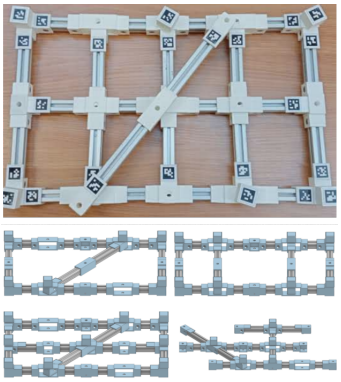
2.4.1 Mechanical Parts Dataset

The first dataset containing mechanical parts is the Actual Artifacts Dataset (AAD) discussed in Bespalov et al. [13]. It consists of four subdatasets where the first three are all

(a) Toy plane assembly from the YCB dataset (reproduced from [17]. © 2015 IEEE).



(b) Three variants of the Belt drive assembly used during the WRS 2018 (reproduced from [124]. © 2019 The Author(s), Taylor & Francis Group. CC BY-NC-ND 4.0[3]).



(c) The RAMP benchmark with four target assembly states (reproduced from [24]. © 2023 IEEE).

Figure 2.8: Examples of robotic assembly benchmarks: (a) Toy plane assembly from the Yale-CMU-Berkeley (YCB) dataset [17], (b) Belt drive assembly variants from the World Robot Summit (WRS) 2018 [124], and (c) Robotic Assembly Manipulation and Planning (RAMP) benchmark with target assembly states [24].

derived from the National Design Repository Dataset [95]. The Manufacturing Classification Dataset comprises 110 models categorized by their manufacturing process, including 56 prismatic machined parts and 54 cast-then-machined parts. The Functional Classification Dataset contains 70 models that have been manually classified according to their functionality, such as being a bracket, a gear, or a housing. To create the Variable Fidelity Dataset, the authors selected 40 models and reduced their polygon count twice by merging surfaces with similar normals, resulting in a total of 120 models. This was done to test the robustness of retrieval algorithms against varying levels of geometric detail. The final dataset, the LEGO Dataset, contains all 47 pieces from the LEGO Mindstorms robotics kit.

Two recent large-scale datasets created for evaluating the retrieval of mechanical parts are the Mechanical Components Benchmark (MCB) collected by Kim et al. [64] and the A Big CAD Model Dataset (ABC) introduced by Koch et al. [69]. MCB contains around 58700 models across 68 categories that were collected from three online databases: TraceParts[4], 3D Warehouse[5], and GrabCAD[6]. Human experts used a web-based tool to filter and annotate the parts. In contrast, the ABC dataset contains over 1 million CAD models without any category annotations. To create it, the authors collected models from Onshape's online platform[7].

[3] https://creativecommons.org/licenses/by-nc-nd/4.0/
[4] https://www.traceparts.com
[5] https://3dwarehouse.sketchup.com
[6] https://grabcad.com
[7] https://www.onshape.com

The datasets discussed so far contained individual mechanical parts. To address the lack of publicly available assembly datasets suitable for testing retrieval methods, Lupinetti et al. [82] introduced a dataset containing 137 CAD assembly models. These models are categorized into 11 classes, such as coupling flanges, differentials, turbines, ... Overall, the dataset comprises approximately 12800 parts, of which 4900 are unique. On average, each model contains about 93 parts, with approximately 36 unique parts per model. Another one is the Fusion 360 Gallery Assembly Dataset collected by Willis et al. [117]. As its name suggests, it aggregates designs from the Autodesk Online Gallery[8] that were created using Autodesk Fusion 360. It contains approximately 8300 assemblies, comprising 154000 individual parts. That is, on average, each assembly consists of 19 parts.

2.4.2 Assembly Benchmarks

The Cranfield Assembly Benchmark, introduced by Collins et al. [25], is one of the earliest benchmarks for long-horizon robotic assembly. The original benchmark includes 17 parts, while a simplified version, which excludes screws, consists of 10 parts [99]. When fully assembled, the parts form a pendulum.

One benchmark suite that contains both short-horizon and long-horizon robot manipulation tasks is the Yale-CMU-Berkeley (YCB) object and model set, described by Calli et al. [17]. It consists of a diverse collection of objects categorized into food items, kitchen items, tools, shape items, and task items. One of them is the toy plane assembly shown in Figure 2.8a, which was used by Neves and Neto [89] for evaluating their approach as described in Section 2.3.2.

Kimble et al. [65] gave an overview of various benchmarking protocols for robotic assembly that have been used in multiple robotics competitions. In detail, the authors describe three task boards of increasing complexity: the first one focuses on basic operations such as peg-in-hole insertions or gear meshing. The second task board introduces more complex tasks, including the handling of flexible parts such as belt tension mechanisms. Finally, the third task board emphasizes the manipulation of cables. A belt drive assembly similar to the one used in the second task board was employed during the World Robot Summit 2018 competition [124]. Three variations are shown in Figure 2.8b. Furthermore, Kiyokawa et al. [68] used this assembly and its variants for evaluating their ASP approach.

The IKEA Furniture Assembly Environment presented in Lee et al. [79] is a virtual assembly benchmark where one or more robotic manipulators must assemble furniture. It features a diverse range of 61 furniture models, including chairs, tables, and various storage units. These models were created based on official IKEA manuals and have between 2 and 15 parts.

Inspired by real-world industrial assemblies, Collins et al. [24] designed the Robotic Assembly Manipulation and Planning (RAMP) benchmark to assess long-horizon robotic assembly tasks. The benchmark consists of nine 20 $\times$ 20 mm aluminum profiles that a robotic manipulator can assemble. To this end, the authors designed specific 3D-printed joints that can be connected using up to 15 pegs. Based on these parts, the authors constructed nine different assembly tasks, which they categorize into easy, medium, and hard

[8] `https://gallery.autodesk.com`

difficulty levels. Four of these tasks are shown in Figure 2.8c. The easy class contains assemblies that require 3-4 peg insertions, while the medium class involves 4-8 peg insertions and more sophisticated skills, such as angled beam insertions. The hard class presents free-form designs with longer assembly sequences.

2.4.3 Discussion

We provided an overview of datasets comprising individual mechanical parts as well as complete assemblies. As previously mentioned, there are some large datasets containing individual mechanical parts. However, the number of datasets consisting of complete assemblies is limited. One reason for this is that high-quality assembly models are often proprietary and closely guarded by the companies that produce them. Another limitation of using publicly available assembly models for ASP is that they frequently require additional processing to address issues such as overlapping or missing parts. To handle overlapping parts, various approaches have been suggested. One method involves shrinking parts to eliminate intersections [49]. An alternative approach is to push overlapping parts away from each other. In fact, as we have discussed in Section 2.2.3, this latter technique was employed by Tian et al. [107] to accelerate removal tests in their ASP framework. Another limitation of these datasets is the lack of information about the properties of their assemblies, such as the number of hands or the specific tools required for assembly. We discussed various assembly properties in Section 2.2. As a case in point, Tian et al. [107] only focused on finding feasible removal paths while ignoring stability constraints.

We also examined specific robotic assembly benchmarks that come with detailed assembly instructions. However, these benchmarks often offer either no variants or few variants with only minor differences between them. Consequently, they are not well-suited for training deep learning approaches.

As shown in Tables 2.1 and 2.2, another approach is to generate assemblies with known properties, for example, out of wooden blocks [41, 39], trusses [47], and aluminum profiles [97, 10]. Inspired by these works as well as the Soma cubes used by Kitz and Thomas [67] and the aluminum profile benchmark described by Collins et al. [24], we propose two assembly generators. The first generator produces aluminum profile assemblies that can be assembled by a single robotic manipulator. Importantly, in contrast to previous work [97, 10, 24] that focused on 2D assemblies (i.e., assembled on a plane such as a table), our generator creates 3D assemblies. The second generator creates Soma cubes of arbitrary size, which have the property that all parts can be removed along straight trajectories.

3 Learning Feasible Assembly Actions

One of the visions of Industry 4.0 is a shift from traditional mass production to mass customization. This transition requires manufacturing systems capable of efficiently producing a wide variety of product variants, often in small batches or even as single units.

One important step during the production process is ASP, i.e., planning a feasible sequence in which the individual parts of a product can be assembled. Traditionally, this step is performed by human experts. However, given the time required to manually plan assembly sequences, it is crucial to automate this step to efficiently handle a large number of product variants.

In fact, as we have shown in Section 2.2.3, automatically deriving assembly sequences from CAD models is an important research area. In particular, we provided an overview of several approaches that utilize AbD to plan a feasible assembly sequence. These methods begin with a fully assembled product and simulate its disassembly by sequentially removing individual parts. The resulting disassembly sequence is then inverted to obtain a feasible assembly sequence.

All of these approaches focused on how to efficiently test whether a part can be removed from the assembly at each step. However, they all face the issue of computational explosion. For an assembly consisting of N parts, they need to conduct $N + (N-1) + (N-2) + \cdots \in \mathcal{O}(N^2)$ tests. This means that even if individual tests are fast, the total number of tests can become overwhelming for assemblies with many parts. Moreover,

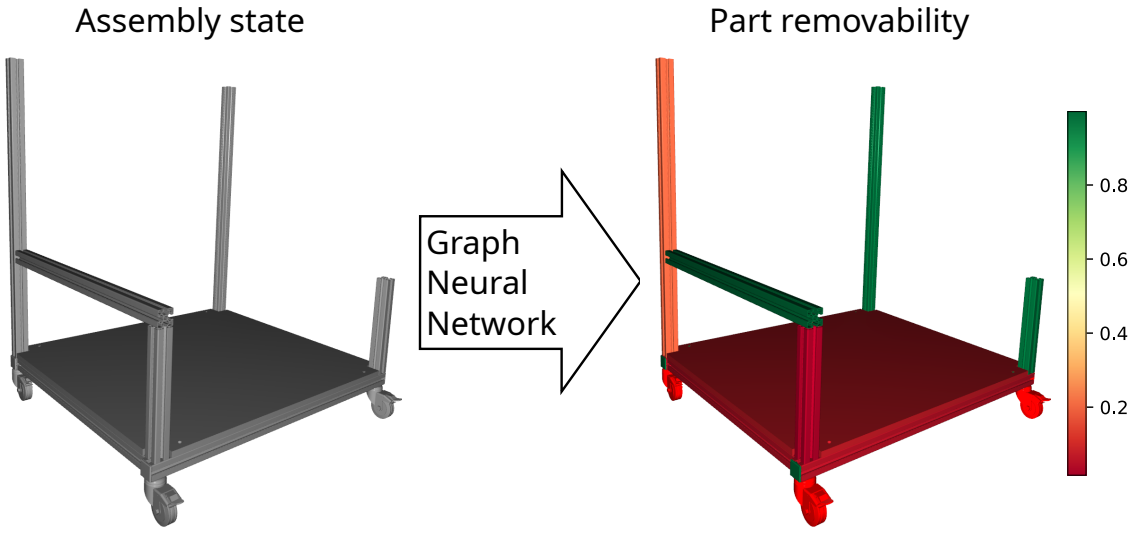


Figure 3.1: The goal of our method is to predict the removability of each part in an assembly (reproduced from Cebulla et al. [19]. © 2023, IEEE).

many of these tests may be unnecessary, as they check for the removability of parts that are clearly not feasible candidates at certain stages of the disassembly process.

As depicted in Figure 3.1, our goal is to reduce the number of unsuccessful tests by learning to predict part removability. This approach is based on two insights: first, product variants often share subassemblies, and second, a part's removability primarily depends on its type and the configuration of surrounding parts. To capture this information, we have developed a graph-based assembly representation where nodes represent individual parts and edges represent connections between parts. Both parts and their connections are represented using 3D shape descriptors [9, 62] derived from the assembly's CAD model. This representation serves as input for a GNN, which predicts the likelihood of successful part removal. We then incorporate these predictions into the AbD process and test parts with higher predicted removability before those with lower removability.

To evaluate our system, we conducted experiments using five real-world assemblies consisting of 29 to 68 parts. We compared our method with a traditional AbD approach proposed by Ebinger et al. [38], with and without an additional bookkeeping heuristic suggested by Dorn et al. [33], to assess the accuracy of removability predictions as well as the overall efficiency gain in the ASP process. Our results demonstrate an accuracy of over 85% in predicting part removability. Furthermore, our approach reduced the number of required removal tests by 64% to 90% compared to the traditional AbD method, and by 30% to 87% when compared to AbD utilizing the bookkeeping heuristic.

In the following sections, we will first position our approach within related literature. Afterwards, we give a formal definition of the feasible ASP problem. Next, we describe a traditional approach to ASP proposed by [38]. We then explain our graph-based assembly representation as well as the architecture of the GNN we used. Next, we elaborate on how we used its predictions to accelerate the AbD process. Finally, we will present experimental results demonstrating the effectiveness of our approach compared to traditional AbD.

The work in this chapter has been published in Cebulla et al. [19].

3.1 Positioning Within Related Literature

We provided a comprehensive review of deep learning approaches for directly predicting feasible assembly sequences in Section 2.2.4, as summarized in Table 2.1. As discussed in Section 2.2.5, these approaches have several key limitations:

1. They lack recovery strategies for handling incorrect predictions, which limits their robustness when encountering errors during sequence generation.
2. They either require prior knowledge of part types [10, 41, 83] or retraining on the evaluation set [125].
3. They generate sequences in the forward direction, necessitating modeling of the complete assembly state at each prediction step. This includes both the currently assembled parts and all parts that remain to be assembled.

Our proposed approach directly addresses these points:

1. Instead of solely relying on the predictions to choose the next part to disassemble, we use a graph search to organize the AbD process. This allows for recovery and backtracking if a prediction turns out to be incorrect.
2. By utilizing a graph search as well as 3D shape descriptors [9, 62] to represent parts and their connections, our approach can handle previously unseen parts.
3. Because we predict the assembly sequence in backward (AbD) direction, the assembly state only contains parts that are still attached to the assembly. In contrast, already removed parts can be ignored.

3.1.1 Bookkeeping Heuristic for AbD

Another approach to reduce unnecessary removal attempts during the AbD process is the bookkeeping heuristic proposed by Dorn et al. [33]. This method maintains a record of parts whose removal was blocked by collisions during earlier attempts. That is, when a part fails a removal test, the colliding parts are recorded. In subsequent disassembly steps, this part is only retested if at least one of its recorded colliding parts has been removed.

While the bookkeeping heuristic reactively avoids retesting parts known to be unremovable, our method proactively predicts which parts are likely to be removable. Thus, it can potentially avoid unnecessary tests before they occur. As we will show in our evaluation, combining our approach with the bookkeeping heuristic can lead to even greater efficiency gains in the AbD process.

3.2 Problem Statement

An assembly, as defined in Section 2.1, is a tuple $\mathcal{A} := (\mathcal{P}, \mathcal{C})$, where $\mathcal{P} = \{p_1, p_2, \ldots, p_N\}$ is the set of its N parts, and $\mathcal{C}$ is the set of relationships between these parts. Each part p_i is characterized by its pose and geometric shape.

We employ the AbD approach to ASP and represent a disassembly sequence as a state-action sequence:

$$u = \{(s_0, a_0), (s_1, a_1), \ldots, (s_{N-1}, a_{N-1})\}, \tag{3.1}$$

where s_t is the assembly state at timestep t, that is, s_0 corresponds to the original assembly $\mathcal{A}$ and s_{N-1} contains a single removable part. Furthermore, a_t represents the disassembly action at timestep t and corresponds to the index of the part to be removed. We define $\mathcal{U}$ as the set of all disassembly sequences, i.e., $u \in \mathcal{U}$.

To describe our problem formally, we introduce a feasibility function $f_f(u) : \mathcal{U} \to \{0, 1\}$ and a removability function $g_f(s_t, a_t) : \mathcal{S} \times \mathcal{A} \to \{0, 1\}$, where the former is defined over the whole disassembly sequence u and the latter over individual state-actions pairs in a sequence:

$$f_f(u) = \begin{cases} 1, & \text{if } u \text{ is feasible,} \\ 0, & \text{otherwise} \end{cases} \tag{3.2}$$

and

$$g_f(s_t, a_t) = \begin{cases} 1, & \text{if } p_{a_t} \text{ is removable in } s_t, \\ 0, & \text{otherwise.} \end{cases} \tag{3.3}$$

We define a sequence u to be feasible if and only if all of its individual disassembly actions were successful.

Our problem is then to find a disassembly sequence $u \in \mathcal{U}$ such that

$$f_f(u) = 1 \iff g_f(s_t, a_t) = 1 \quad \forall t = 0, 1, \ldots, N-1. \tag{3.4}$$

3.3 Planning Feasible Assembly Sequence via AbD

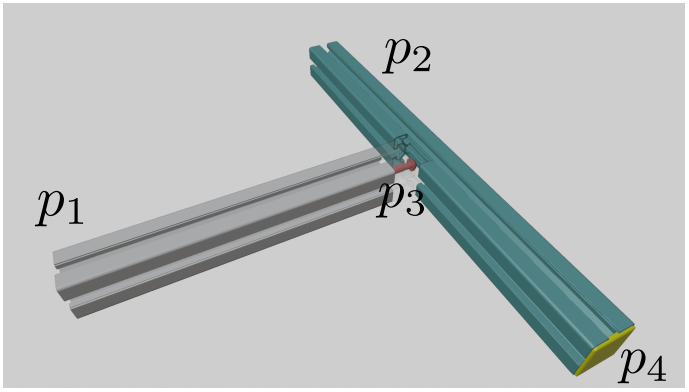
One approach to finding a feasible assembly sequence is to organize the AbD process as a graph search through a directed, acyclic disassembly graph (DG), where the root of the graph contains the fully assembled product and intermediate nodes represent valid subassemblies. As the graph search progresses, the DG expands. That is, to expand a node, the framework iteratively tests the removability of parts in the current disassembly state. If the removal of a part is feasible, a new child node is created representing the disassembly state after the part's removal. This continues until a leaf node is reached, which represents a fully disassembled state. Consequently, each path from the root to a leaf node in the DG represents a valid disassembly sequence.

To more efficiently test for removability, we follow the suggestion of Ebinger et al. [38] and employ a two-step testing approach. We first use a fast test, and if it fails, we then use a more comprehensive, slower one. Specifically, the fast test checks if a part can be removed along a set of predefined directions, which correspond to the clustered surface normals of all parts in the assembly. If the fast test fails for all parts in the current disassembly state, we perform a more thorough test using RRT [75], a sampling-based motion planning algorithm, to search for a feasible removal path.

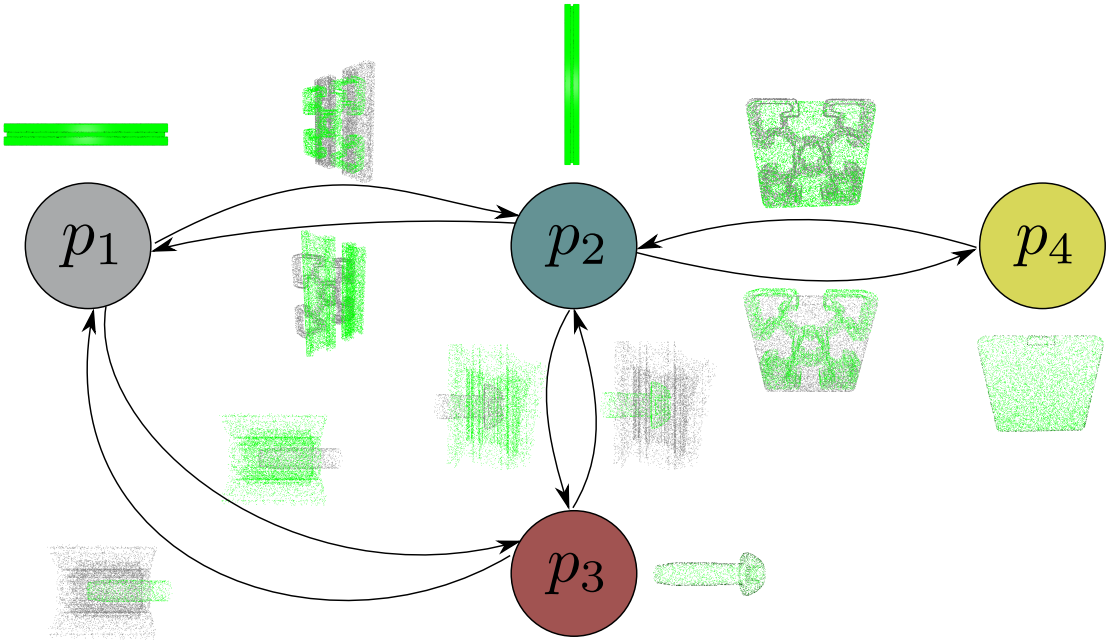
While Ebinger et al. [38] solely focused on removability, we also tested for stability. Specifically, we added a simple gravity constraint based on the assumption that each planned assembly sequence will be contact-coherent as discussed in Section 2.1.4, i.e., each newly added part makes contact with at least one previously assembled part. An assembly state is considered stable only when all parts are linked, whether directly or indirectly through physically connected parts, to at least one part belonging to the set of fixated parts upon which the assembly is built.

The framework can be used with various graph search algorithms, and DFS and BFS were used by Ebinger et al. [38]. Because we are interested in efficiently finding a feasible assembly sequence, we used DFS.

An important aspect of the framework that significantly impacts its efficiency is the order in which parts are tested for removal within each disassembly state. Ebinger et al. [38] used an arbitrary order. However, this approach can lead to unnecessary computational overhead.



(a) A 3D assembly model consisting of four parts.



(b) Graph representation of the assembly model with extracted point clouds on the edges.

Figure 3.2: Depiction of a four-part 3D assembly model and its corresponding graph representation (reproduced from Cebulla et al. [19]. © 2023, IEEE). While the extracted point clouds are shown, the computation of 3D shape descriptors from these point clouds is not displayed.

3.4 Graph-Based Assembly Representation

The main idea behind our approach is to predict whether parts can be removed in a certain assembly state and then test parts with a high removability probability first. We assume that a part's removability is primarily determined by two local factors: its function, which is closely correlated with its shape, and its immediate connections to other parts. For instance, a screw's function is to fasten different parts together, and its shape reflects this purpose. Consequently, any parts held by the screw cannot be removed until the screw itself is removed. By focusing on these local characteristics rather than the entire assembly structure, we not only capture the most relevant information for removability but also address potential scalability issues that may arise. To effectively capture these factors, we propose a graph-based representation. This allows us to represent both the

individual properties of parts and their connections to other parts in a format that can be provided to a GNN.

3.4.1 Graph Structure

Given an assembly $\mathcal{A}$ as previously defined, we need to capture the neighborhood between its parts. Therefore, we define $\mathcal{C}_l$, the set of relationships, more precisely, to contain all physical connections between the parts in $\mathcal{P}$. A physical connection exists between two parts if their surfaces are within a small threshold distance of each other.

Formally, let $\mathcal{Q}_{p_i}$ be the set of all points on the surface of part p_i. We define:

$$\mathcal{C}_l := \{(p_i, p_j) | p_i, p_j \in \mathcal{P}, \min_{q_i \in \mathcal{Q}_{p_i}, q_j \in \mathcal{Q}_{p_j}} ||q_i - q_j||^2 < \epsilon_c\}, \tag{3.5}$$

where $\epsilon_c > 0$ is a small threshold distance.

We can now represent $\mathcal{A}$ as a directed graph $G_\mathcal{A} = (N_\mathcal{A}, E_\mathcal{A})$, where $N_\mathcal{A}$ is the set of nodes and $E_\mathcal{A}$ is the set of edges between them.

Each node $n_i \in N_\mathcal{A}$ corresponds to a part $p_i \in \mathcal{P}$ and each directed edge $e_{i,j}$ to a connection between parts p_i and p_j. Formally,

$$N_\mathcal{A} := \{n_i | p_i \in \mathcal{P}\} \tag{3.6}$$

$$E_\mathcal{A} := \{e_{i,j} | (p_i, p_j) \in \mathcal{C}_l\} \tag{3.7}$$

3.4.2 Node and Edge Attributes

As stated above, we assume that the removability of a part is mainly dependent on its function (as defined by its shape) and its neighborhood. Importantly, this function is independent of the part's size, orientation, and location within the assembly. While the edges in our graph already capture the connections between parts, we want to include specific information about the exact regions where parts make contact. To achieve this, we represent the point clouds of individual parts and those of the contact regions between parts using 3D shape descriptors [62, 9]. In particular, we compute the spherical harmonic representation [62] of their 3D shape histograms [9], ensuring invariance to scale, rotation, and translation. Figure 3.2 provides an example of our graph-based assembly representation before the computation of the spherical harmonic representation.

Spherical Harmonic Representation of 3D Shape Histograms

Spherical harmonics [62] are a set of orthogonal functions defined on the surface of a sphere. They can be used to approximate any function on a sphere, similar to how Fourier series approximate functions on a circle. In our context, we use them to represent a point cloud in a way that is invariant to rotation.

In detail, given a point cloud $\mathcal{Q}$, we compute a spherical harmonic shape descriptor as follows:

1. First, to be scale and translation invariant, we normalize $\mathcal{Q}$ by centering it at the origin and scaling it to fit within a unit sphere.
2. We then express its points in spherical coordinates (r, θ, ϕ).
3. For the next step, we calculate the 3D histogram for these points, where each bin defines a range for r, θ, ϕ.
4. If we fix a range for r, then all the bins for this range form a discrete spherical function $f(\theta, \phi)$.
5. We now approximate this function by the sum of the spherical harmonic basis functions $Y_l^m(\theta, \phi)$ up to degree L:

$$f(\theta, \phi) = \sum_{l=0}^{L} f_l(\theta, \phi) \tag{3.8}$$

$$f_l(\theta, \phi) = \sum_{m=-l}^{l} a_l^m Y_l^m(\theta, \phi) \tag{3.9}$$

   where f_l are the frequency components of f and a_l^m are the coefficients obtained by projecting $f(\theta, \phi)$ onto $Y_l^m(\theta, \phi)$.
6. A property of spherical functions is that their L2-norm is not affected by rotation. We use this to achieve rotation invariance. Specifically, to construct the shape descriptor $\vec{h} \in \mathbb{R}^D$, we compute the L2-norm of the coefficients for every radius bin and then concatenate them

$$\vec{h} = \left(\|f_0(\theta, \phi)\|_2 , \|f_1(\theta, \phi)\|_2 , \ldots , \|f_L(\theta, \phi)\|_2 \right). \tag{3.10}$$

Node Attributes

For each node $n_i \in N_{\mathcal{A}}$, we compute a node attribute vector $a_i^n \in \mathbb{R}^D$ to represent the shape of the corresponding part p_i. That is, we use the spherical harmonic representation of the 3D shape histograms computed for the point cloud $\mathcal{Q}_i$ of part p_i

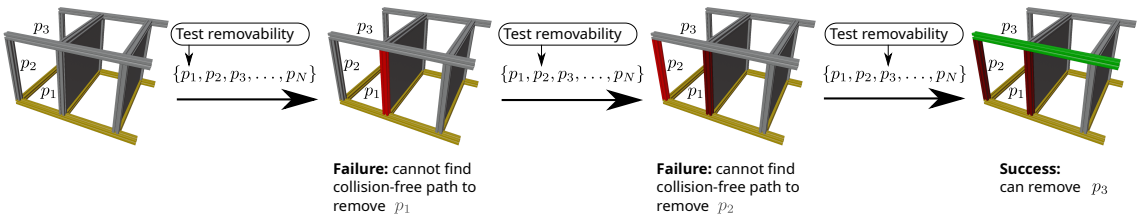
$$a_i^n := \vec{h}_{p_i} \in \mathbb{R}^D \tag{3.11}$$

Edge Attributes

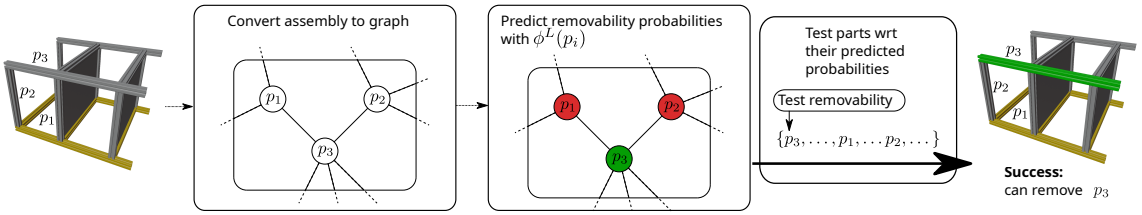
For each edge $e_{i,j} \in E_{\mathcal{A}}$, we compute an edge attribute vector $a_{i,j}^e \in \mathbb{R}^D$ that captures the characteristics of the connection between parts p_i and p_j. That is, we first extract the point cloud $\mathcal{Q}_i^j = \{q_i \in \mathcal{Q}_{p_i} | \exists q_j \in \mathcal{Q}_{p_j} : ||q_i - q_j|| < \epsilon\} \subseteq \mathcal{Q}_{p_i}$ which contains all points in $\mathcal{Q}_{p_i}$ that are closer than a threshold ϵ to any point in $\mathcal{Q}_{p_j}$. We then compute the edge attribute as

$$a_{i,j}^e := [\vec{h}_{j,i}, \vec{h}_{i,j}] \in \mathbb{R}^{2D}, \tag{3.12}$$

where $\vec{h}_{i,j}$ and $\vec{h}_{j,i}$ are the shape descriptors computed for $\mathcal{Q}_j^i$ and $\mathcal{Q}_i^j$, respectively.



(a) Vanilla AbD: Test the removability of assembly parts $\mathcal{A} := \{p_1, p_2, p_3, \ldots, p_N\}$ in an arbitrarily predefined order.



(b) Our approach: Test parts in order corresponding to their predicted removability ϕ^L.

Figure 3.3: A comparison between the **vanilla AbD** method and **our approach** (both figures are reproduced from Cebulla et al. [19]. © 2023, IEEE). The parts shown in yellow are fixed in the xy-plane, and serve as a stable base for the other parts. Gravity acts in the -z-direction.

3.5 Learning and Predicting Feasible Assembly Actions

The vanilla AbD framework's arbitrary part testing order is inefficient, as it might attempt to remove many non-removable parts before finally testing a part that can actually be removed. To address this limitation, we propose a novel method that uses a GNN to predict the removability of individual parts. It then prioritizes parts with higher predicted removability, potentially reducing unnecessary removal attempts. Figure 3.3 shows a comparison between the vanilla approach proposed by Ebinger et al. [38] and ours. Similar to Atad et al. [10] and Ma et al. [83], we train the GNN via supervised learning, in contrast to the RL methods used by Zhou et al. [127] and Ghasemipour et al. [41].

In the following sections, we first elaborate on our choice of the supervised learning regime. We then introduce our GNN architecture and explain how we integrate its predictions into the AbD framework.

3.5.1 Supervised Learning vs Reinforcement Learning

It is straightforward to interpret the disassembly sequence $u \in \mathcal{U}$ as a rollout of an MDP and then, as explained in Section 2.2.4, apply RL like DQL to generate feasible sequences. In fact, this approach was taken by Zhou et al. [127] and Ghasemipour et al. [41].

However, the goal of RL is to learn a policy that maximizes the expected cumulative reward, as defined in Equation 2.2. This means it considers not only the immediate effects of an action but also its impact on future states. When planning the assembly sequence in a forward direction, it is necessary to guarantee geometric feasibility. For example, if an assembly consists of a housing with internal components, assembling the housing first

would block the insertion of the internal parts. This problem does not arise if backward (AbD) ASP is used.

A similar argument can also be made for considering a stability constraint. For example, assembling top-heavy parts before establishing a solid foundation could lead to an unstable assembly that might collapse. In this case, establishing this foundation later on might not be possible without colliding with already assembled parts. In contrast, this problem does not occur with backward (AbD) ASP, as disassembling a stable assembly typically maintains stability throughout the process. Independent of the assembly direction, it is possible to get into states that are less stable than other states, which, however, cannot be distinguished by a binary stability constraint.

3.5.2 The GNN Architecture

A GNN is a neural network architecture specialized in processing graphs with a varying number of nodes and edges [12]. It consists of multiple layers with shared weights, where each layer updates the graph's node and edge attributes based on their local neighborhood. Formally:

$$G^{(l+1)} = \Phi^l \left(G^l \right), \forall l \in [0, L], \tag{3.13}$$

where Φ^l is the l-th GNN layer, G^0 the original, input graph, G^l for $l \in [1, \ldots, L]$ its intermediate representations and G^{L+1} its final representation, i.e., the GNN's output.

In our implementation, we applied a linear transformation to the edge attributes in the first layer, ensuring their dimensionality matches that of the node attributes. The update process in each layer l consists of two main steps: edge updates and node updates.

1. For the edge update, we use:

$$a_{e_{i,j}}^{(l+1)} = \phi_e^l \left(a_{n_i}^l + a_{e_{i,j}}^l + a_{n_j}^l \right) \tag{3.14}$$

   where $a_{n_i}^l$ and $a_{n_j}^l$ are the attributes of nodes n_i and n_j at layer l, respectively, $a_{e_{i,j}}^l$ is the attribute of edge $e_{(i,j)}$ at layer l and ϕ_e^l is an FNN.

2. The node update is performed in two steps:

   a) Aggregation of neighborhood information:

$$\bar{a}_{n_i}^{(l+1)} = \sigma^l \left(ReLU \left(a_{n_i}^l + a_{e_{i,j}}^{(l+1)} \right) + \epsilon | e_{i,j} \in E_{\mathcal{A}} \right) \tag{3.15}$$

   where ϵ is a small constant for numerical stability and σ^l is a learned *SoftMax* aggregator that outperformed other functions like *mean()* or *max()* as demonstrated in [80].

   b) Update of node attributes:

$$a_{n_i}^{(l+1)} = \phi_n^l \left(a_{n_i}^l + \bar{a}_{n_i}^{(l+1)} \right) \tag{3.16}$$

   where ϕ_n^l is another FNN.

We used two layers with 64 neurons and ReLU activation functions to implement the two FNNs ϕ_e^l and ϕ_n^l. The same two networks are used in every layer Φ^l.

In the final layer Φ^L, we used a sigmoid activation function to compute a single output between 0 and 1 for each node n_i, which we interpret as the removability of the corresponding part p_i.

3.5.3 Reordering Parts Based on Their Predicted Removability

To speed up the AbD process, we first use our trained GNN to predict part removability. Then, we prioritize testing parts that are more likely to be removable, thereby potentially reducing the number of unnecessary removal attempts. In detail, the process, also illustrated in Figure 3.3b, is as follows:

1. Utilizing the procedure described in Section 3.4, we first represent any new assembly $\mathcal{A}$ as a graph $G_{\mathcal{A}} = \{N_{\mathcal{A}}, E_{\mathcal{A}}\}$.
2. We then use the trained GNN Φ to predict the removability probability n_i^L for each node $n_i \in N_{\mathcal{A}}$.
3. The parts p_i corresponding to the nodes $n_i \in N_{\mathcal{A}}$ are then ordered from highest predicted removability to lowest predicted removability.
4. Utilizing this ordering, we iteratively test parts until finding one that can be removed.
5. The removal of a part results in a new subassembly, which is then represented as a new graph.

We iteratively apply these steps to each resulting subassembly until we obtain a complete disassembly sequence.

3.6 Evaluation

To evaluate our approach, we conducted two sets of experiments:

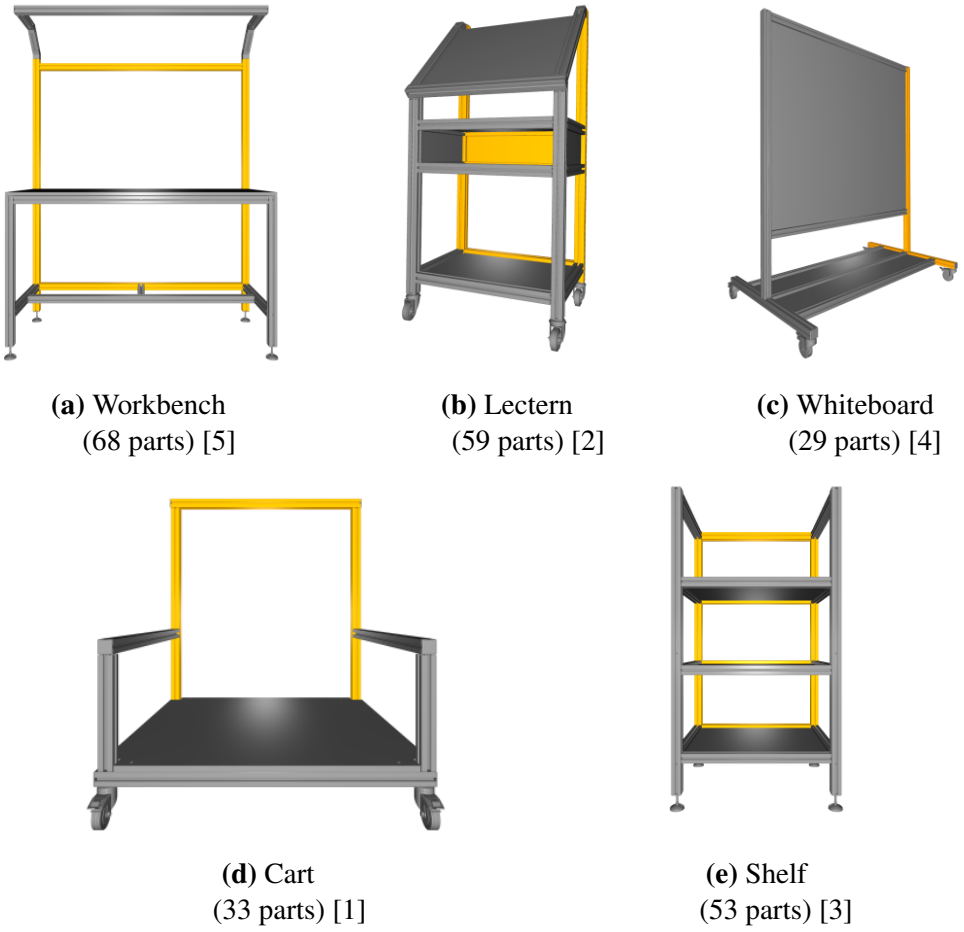
1. We analyzed how suitable our proposed graph representation is for inferring part removability.
2. We compared the efficiency of our approach with the vanilla AbD approach proposed by Ebinger et al. [38], both with and without utilizing the bookkeeping heuristic introduced by Dorn et al. [33]. We briefly described its function in Section 3.1.1. Moreover, we also investigated whether our approach can be improved by combining it with this bookkeeping heuristic.

In the following sections, we first describe the dataset used for both experiments and how we trained our GNN on it. We then discuss each experiment and its results in detail.

3.6.1 Dataset

To evaluate our approach and train the GNN, we created a dataset using five real-world assemblies illustrated in Figure 3.4: a workbench (68 parts), a lectern (59 parts), a whiteboard (29 parts), a cart (33 parts), and a shelf (53 parts).

For each assembly, we generated training data by running the vanilla AbD framework 20 times, starting from the fully assembled state and continuing until complete disassembly was achieved. At each step of the disassembly process, we recorded the current assembly state $\mathcal{A}_i$ as well as the removability label l_i for each part in the current state (1 if removable, 0 if not removable). This resulted in the dataset $\mathcal{D} = \{(\mathcal{A}_i, l_i) | i \in 0, 1, \ldots, N\}$.



(a) Workbench (68 parts) [5]

(b) Lectern (59 parts) [2]

(c) Whiteboard (29 parts) [4]

(d) Cart (33 parts) [1]

(e) Shelf (53 parts) [3]

Figure 3.4: The real-world assemblies used for evaluation, which we obtained and cleaned from the given source. To have a stable base for each assembly, we fixed some of the parts (marked in yellow) in the xy-plane. They are excluded from the part count (all figures are reproduced from Cebulla et al. [19]. © 2023, IEEE).

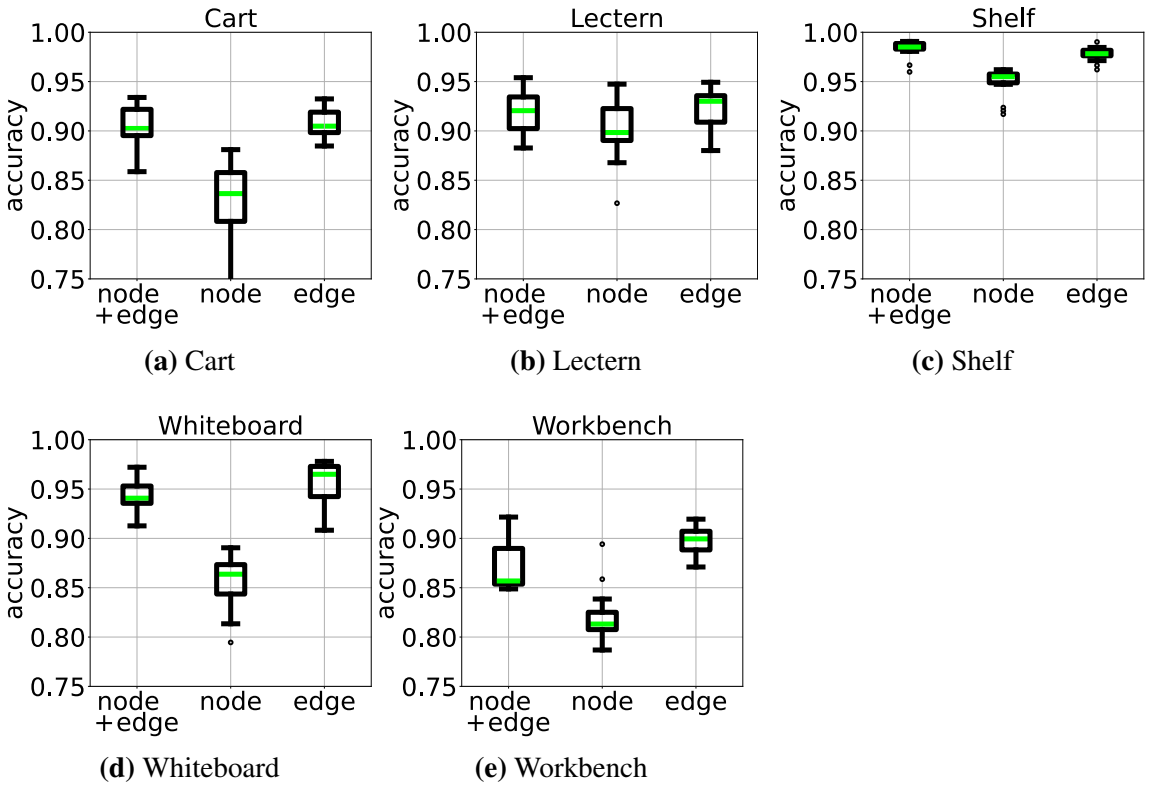
3.6.2 Training the GNN

We trained the GNN on a dataset $\mathcal{D} = \{(\mathcal{A}_i, l_i) | i \in \{0, 1, \ldots, N\}\}$ collected as described in the previous section, where we used a binary cross-entropy loss and the ADAM optimizer [66].

3.6.3 Analyzing the Graph Representation

To evaluate the effectiveness of our proposed graph representation for inferring part removability, we conducted a series of experiments using different combinations of node and edge attributes. We used a five-fold cross-validation, where for each assembly, we trained a GNN for 50 epochs on data from the other four assemblies and tested it on the target assembly. We repeated this 10 times for each assembly to account for variability and then measured the averaged prediction accuracy *acc*, where

$$acc = \frac{\text{number of correct predictions}}{\text{total number of samples}}.$$



(a) Cart **(b)** Lectern **(c)** Shelf

(d) Whiteboard **(e)** Workbench

Figure 3.5: Boxplots showing the prediction accuracy for different assemblies (results published in Cebulla et al. [19]. © 2023, IEEE).

We considered a part to be removable if its predicted removability was greater than 0.5.

We tested three different configurations:

1. **Only node attributes**: using only the 3D shape descriptors of individual parts.
2. **Only edge attributes**: using only the 3D shape descriptors of contact areas between parts.
3. **Combined node + edge attributes**: using part shapes as well as contact area information.

Figure 3.5 shows boxplots of the prediction accuracy for each assembly under these three configurations. Using **only node attributes**, we achieved a median accuracy above 0.8 for all assemblies, which suggests that the shape of a part provides valuable information about its removability. However, **edge attributes alone** consistently outperformed **node attributes**, indicating that the connections between parts are more informative for predicting removability than the shapes of individual parts. This makes intuitive sense, as the removability of a part often depends more on the surrounding parts than on itself. The **combination of node and edge attributes** generally yielded the highest accuracy, although the improvement over **edge attributes alone** was often marginal. This suggests that most of the relevant information for removability prediction is captured in the **edge attributes**, with **node attributes** providing some complementary information.

We observed that prediction accuracy varied across different assemblies; for instance, the shelf and whiteboard assemblies showed higher accuracies and less variability compared to the workbench and cart, suggesting that some assembly structures may be inherently more predictable in terms of part removability.

Table 3.1: The total disassembly time and the number of removal tests required by the AbD framework [38], described in Section 3.3, were evaluated. Performance was compared both without (**Vanilla**) and with (**BK**) the bookkeeping heuristic [33], detailed in Section 3.1.1. These results were further compared with our proposed approach, presented in Section 3.5, which was also tested without (**GNN**) and with (**GNN + BK**) the same heuristic. Each of the five assemblies was evaluated across ten different initial part orderings. The best results for each assembly are highlighted. (results published in Cebulla et al. [19]. © 2023, IEEE).

Assembly	Approach	Time (s) (mean + std. over 10 runs)	Number of removal tests (mean + std. over 10 runs)		
			Total	Failed (collision)	Failed (gravity)
Workbench	**Vanilla**	816 ± 90	1153 ± 107	1032 ± 113	53 ± 16
	BK	439 ± 75	604 ± 46	483 ± 45	53 ± 16
	GNN	310 ± 34	420 ± 47	344 ± 46	**8 ± 2**
	GNN + BK	**224 ± 28**	**281 ± 30**	**205 ± 29**	**8 ± 2**
Lectern	**Vanilla**	591 ± 190	725 ± 74	600 ± 84	66 ± 25
	BK	348 ± 89	499 ± 56	374 ± 61	66 ± 25
	GNN	72 ± 13	123 ± 25	60 ± 21	**4 ± 9**
	GNN + BK	**57 ± 12**	**85 ± 14**	**23 ± 8**	**4 ± 9**
Whiteboard	**Vanilla**	217 ± 115	186 ± 36	144 ± 33	13 ± 7
	BK	184 ± 120	157 ± 25	115 ± 21	13 ± 7
	GNN	**66 ± 32**	**35 ± 5**	**1 ± 1**	**5 ± 5**
	GNN + BK	**66 ± 32**	**35 ± 5**	**1 ± 1**	**5 ± 5**
Cart	**Vanilla**	191 ± 53	252 ± 36	197 ± 40	22 ± 8
	BK	130 ± 33	174 ± 12	119 ± 16	22 ± 8
	GNN	**51 ± 15**	90 ± 11	49 ± 11	**7 ± 2**
	GNN + BK	54 ± 10	**88 ± 10**	**47 ± 10**	**7 ± 2**
Shelf	**Vanilla**	258 ± 49	579 ± 85	477 ± 80	49 ± 19
	BK	185 ± 50	447 ± 79	345 ± 71	49 ± 19
	GNN	35 ± 5	**58 ± 3**	**6 ± 3**	**0 ± 0**
	GNN + BK	**34 ± 6**	**58 ± 3**	**6 ± 3**	**0 ± 0**

3.6.4 Improving ASP by Reordering Parts

To evaluate the effectiveness of our GNN-based approach in improving the efficiency of ASP, we conducted a series of experiments comparing four different methods:

1. **Vanilla**: The vanilla AbD framework [38], presented in Section 3.3, using an arbitrary part testing order.
2. **Bookkeeping (BK)**: The AbD framework enhanced with the bookkeeping heuristic [33], described in Section 3.1.1.
3. **GNN**: Our proposed approach that reorders parts according to their predicted removability.
4. **GNN + BK**: A combination of our GNN-based approach with the bookkeeping heuristic [33].

For the two GNN-based approaches, we trained the GNN using five-fold cross-validation over the five assemblies, as detailed in Section 3.6.3. We ran each method 10 times on each of the five assemblies. Importantly, for each run, we used the same predefined initial part ordering for all four methods, but varied this order across the 10 runs. Furthermore, the same trained GNN is used for both the third and the fourth method. This ensures a fair comparison between methods while also testing their robustness to different initial conditions. We used the full assembly representation, i.e., with node and edge attributes. Table 3.1 presents the results, showing the total disassembly time and the number of removal tests for each method across all assemblies. The method using the **bookkeeping** heuristic consistently reduces the number of failed removal attempts due to collisions compared to the **vanilla** approach. However, because the heuristic only tracks previous collisions, it cannot reduce the number of failed removal attempts where the gravity constraints were violated. Our approach (**GNN**), on the other hand, consistently outperformed both the **vanilla** AbD method and the **bookkeeping** heuristic across all assemblies. Specifically, it reduced the total number of removal attempts by 64% to 90% and achieved speed-ups ranging from 2.6 to 7.4 times faster. When compared to the AbD framework enhanced with a **bookkeeping** heuristic, it reduced removal attempts by 30% to 87% while achieving speed-ups of 1.4 to 5.3 times. Notably, our approach (**GNN**) significantly reduced both collision-based and gravity-based failed removal attempts, which indicates that our GNN successfully learned to predict not only geometric feasibility but also stability constraints. The degree of improvement varied across different assemblies, with the shelf assembly showing the highest improvement and the workbench the least, possibly due to differences in complexity and structure. Interestingly, the combination of our GNN approach and the bookkeeping heuristic (**GNN + BK**) further improved performance for three of the five assemblies. For the workbench assembly, this reduced the number of removal tests by 33% compared to using the **GNN** method alone. The relatively low standard deviations in our results, despite varying initial part orderings across runs, suggest that the **GNN**-based approach provides consistent improvements and is robust to different starting conditions.

4 Generating Assemblies

To accelerate the ASP process using deep learning techniques, high-quality assembly datasets are required for training. Unfortunately, such datasets are often proprietary and closely guarded by companies. While some publicly available assembly datasets exist, they come with their own challenges and often require additional processing. For instance, parts might be overlapping or missing. In the previous chapter, we had to clean our dataset manually by readjusting part poses and adding missing screws and screw holes. As discussed in Section 2.4, other researchers have developed approaches to automatically handle intersecting parts, such as pushing parts out or shrinking them. However, publicly available assembly datasets have another limitation: they often lack information about assembly properties. This includes details such as the number of hands required (which determines the number of fixtures or robotic manipulators needed) or the specific tools necessary for assembly.

Conversely, researchers focusing on specific questions may require assemblies with particular properties. Therefore, a common approach is to generate assemblies, for example, out of wooden blocks [41, 39], trusses [47], and aluminum profiles [97, 10], which have known characteristics. In this chapter, we introduce two assembly generators. The first one produces 3D aluminum profile assemblies such as the one shown in Figure 4.1. It has been published in Cebulla et al. [20]. The second generator creates Soma cubes as illustrated in Figure 4.2.

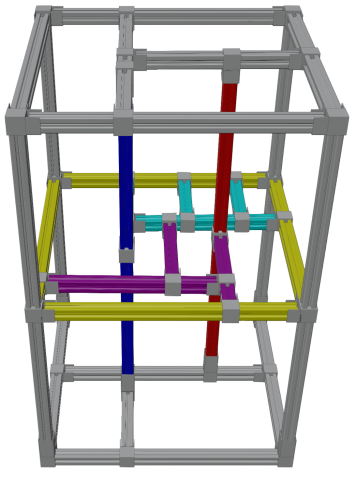


Figure 4.1: A generated aluminum profile assembly (reproduced from Cebulla et al. [20]. © 2024, IEEE).

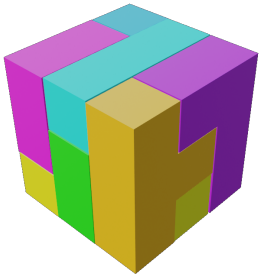
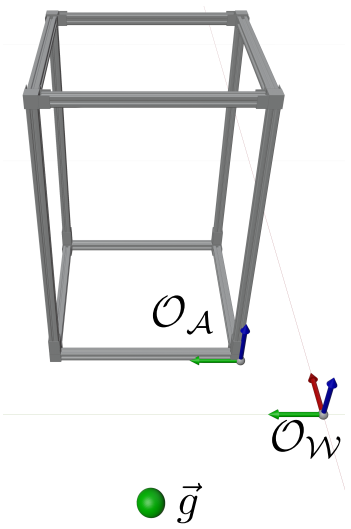
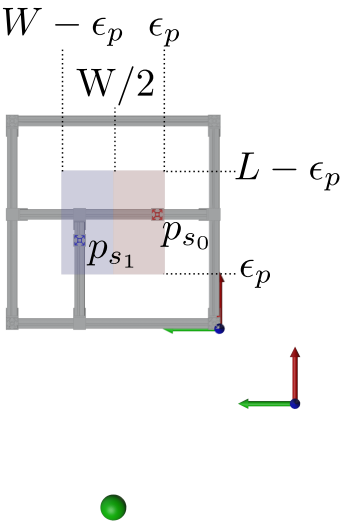


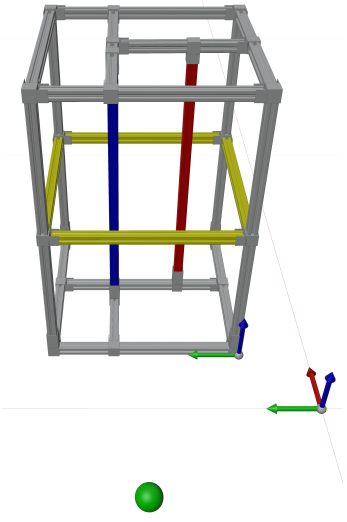
Figure 4.2: A generated Soma cube.



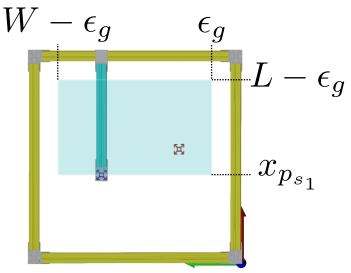
(a) The base structure is composed of 12 profiles.



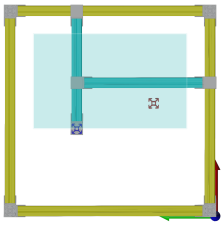
(b) Top view of support profiles p_{s_0} (in red) and p_{s_1} (in blue) being added to the structure.



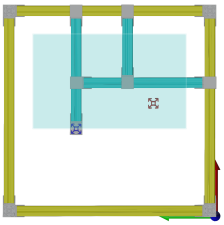
(c) Each layer contains a frame consisting of four yellow profiles.



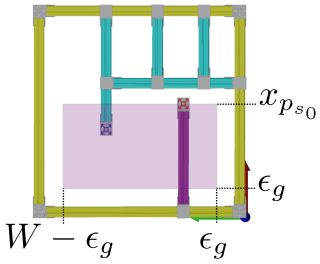
(d) The first back profile is aligned along the x-axis and connected to the support profile p_{s_1}.



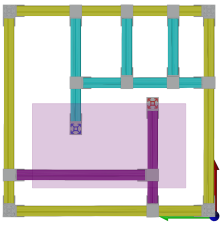
(e) The second back profile is oriented along the y-axis.



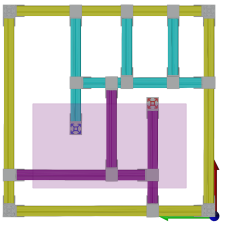
(f) The third and fourth back profiles are both aligned along the x-axis.



(g) The first front profile is aligned along the x-axis and connected to the support profile p_{s_0}.

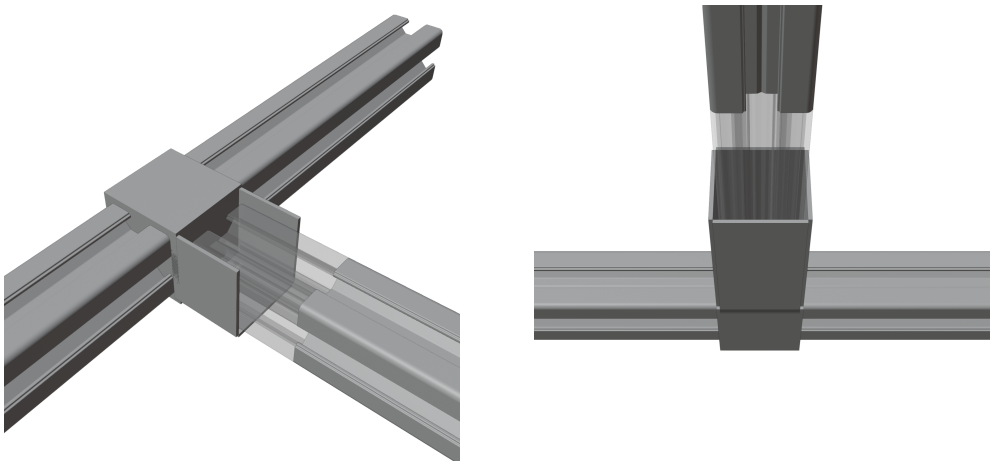


(h) The second front profile is aligned along the y-axis.



(i) The third front profile is aligned along the x-axis.

Figure 4.3: Illustration of the step-by-step creation process of the assembly depicted in Figure 4.1 (all are reproduced from Cebulla et al. [20]. © 2024, IEEE). The top row shows the position $\vec{g}$ from which the profiles are picked up alongside the world coordinate system $\mathcal{O}_\mathcal{W}$ and the assembly coordinate system $\mathcal{O}_\mathcal{A}$. In the subsequent steps, only the assembly coordinate system $\mathcal{O}_\mathcal{A}$ is shown.



(a) The U-shaped connector supports profiles that lie in the xy-plane.

(b) The O-shaped connector supports vertical profiles parallel to the z-axis.

Figure 4.4: An O- and a U-shaped connector (both reproduced from Cebulla et al. [20]. © 2024, IEEE).

4.1 Aluminum Profile Assemblies

Inspired by the assembly benchmark of Collins et al. [24] and building on the work of Rodríguez et al. [97] and Atad et al. [10], we implemented a generator that creates 3D assemblies using 20 mm $\times$ 20 mm aluminum profiles. While previous research focused on 2D assemblies that were constructed in a plane, thereby neglecting stability constraints, we aimed to produce more complex 3D structures. Additionally, we required that these assemblies could be assembled by a single robotic manipulator that has a two-finger gripper as its end-effector.

Because of this requirement, we employ two types of connectors in our assemblies, added wherever one profile's short side contacts another's long side. Profiles parallel to the ground, i.e., those in the xy-plane of the world coordinate system $\mathcal{O}_\mathcal{W}$, use U-shaped connectors, whereas upright profiles, i.e., those along the z-axis, use O-shaped connectors. Both connector types are depicted in Figure 4.4.

Each assembly is generated following a three-step process. First, we establish a base structure that serves as the foundation for all other profiles. Next, we position several supporting profiles within the base structure. Finally, we add multiple layers of profiles, each connected to both the base structure and the supporting profiles. All profiles are parallel to one of the axes of $\mathcal{O}_\mathcal{W}$.

We will now discuss each step in more detail. An overview of these steps is also given in Figure 4.3.

4.1.1 Base Structure

The base structure of each assembly, as shown in Figure 4.3a, is a cuboid consisting of 12 aluminum profiles. Four profiles form a horizontal square at the bottom, and four more create an identical square at the top, with four vertical profiles connecting them at the

corners. Since all other profiles are positioned within this base structure, its height H, width W, and length L define the overall assembly size.

4.1.2 Support Profiles

After establishing the base structure, we add several support profiles. As their name implies, their function is to provide additional support for the layers of profiles that we will add in the next step.

Support profiles are oriented parallel to the z-axis of $\mathcal{O}_{\mathcal{W}}$ and positioned within the base structure. For clarity, we consider two support profiles p_{s_0} and p_{s_1}, shown in Figure 4.3b, though more can be added for increased complexity. Their x-positions are randomly chosen between ϵ_p and $L - \epsilon_p$ and their y-positions between $\epsilon_p < y_{p_{s_0}} < W/2$ and $W/2 < y_{p_{s_1}} < W - \epsilon_p$, where L and W are the assembly length and width, respectively, and ϵ_p is the minimum profile length. This ensures that p_{s_0} is in the left half and p_{s_1} is in the right half. Both profiles connect to the bottom and top layers via additional profiles randomly oriented along either the x- or y-axis.

4.1.3 Adding Profile Layers

With the base structure and support profiles in place, we now add multiple layers of profiles to them. Each one consists of profiles oriented along either the x- or y-axis and is connected to both the base structure and the support profiles.

The creation of each layer follows a three-step process:

Creating a Frame

As illustrated in Figure 4.3c, we create a frame using four profiles in the xy-plane, which are connected to the base structure. It forms the outline of the layer and provides attachment points for subsequent profiles.

Adding Profiles to the Back

Next, we add profiles to the "back" of the layer, i.e., these are all profiles that are partially or completely behind the frontmost support profile when viewed from a predetermined position $\vec{g}$. Here, $\vec{g}$ is the position where profiles are stored and retrieved.

To add a profile, we first randomly determine its orientation either along the x-axis or the y-axis. If an unconnected support profile exists, we randomly decide whether to connect the new profile to it.

- If it should be connected, we set, depending on its orientation, either its x- or y-coordinate accordingly. This is shown in Figure 4.3d.
- In case the new profile should not be connected, or if all support profiles already have connections, we determine the profile's position based on its orientation:

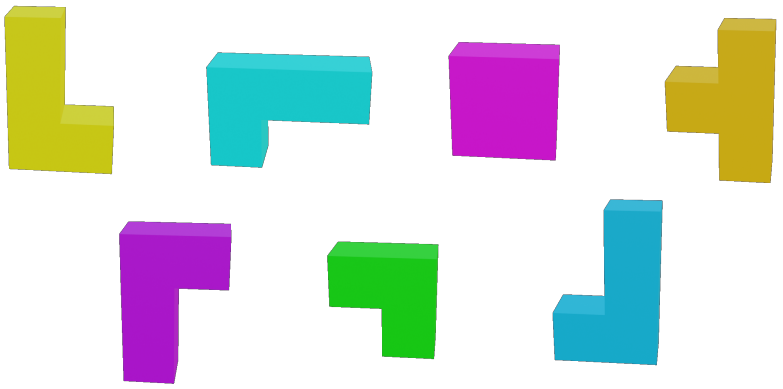


Figure 4.5: Individual parts of the Soma cube shown in Figure 4.2.

- If the profile is oriented along the y-axis: We randomly select its x-coordinate between $x_{p_{s_i}}$ and $L - \epsilon_g$, where $x_{p_{s_i}}$ is the x-coordinate of the support profile closer to the front, L is the length of the assembly, and ϵ_g is the minimum distance required between parallel profiles for robotic grasping. An example is given in Figure 4.3e.

- Otherwise, we choose its y-coordinate between ϵ_g and $W - \epsilon_g$, where W is the width of the assembly. This is illustrated in Figure 4.3f.

After determining the profile's position, we verify that the new profile maintains a minimum distance ϵ_g from all parallel profiles. If it does not, we discard it and restart the process. Otherwise, we add it at full length, which, depending on its orientation, corresponds to either W or L.

If the new profile intersects with any existing profiles, we split it into segments, removing all segments that are either shorter than ϵ_p (the minimum allowable profile length) or are partially or fully in front of the support profile that's closer to the front. We then randomly select one of the remaining segments and add it to the assembly.

Adding Profiles to the Front

Finally, as shown in Figures 4.3g to 4.3i, we add profiles to the "front" of the layer. This step is similar to the previous one, with the following adjustments:

- When determining the position for profiles not connected to support profiles that are oriented along the y-axis, we randomly select its x-coordinate between ϵ_g and $x_{p_{s_j}}$, where $x_{p_{s_j}}$ is the x-coordinate of the support profile that is in the back.

- When splitting intersecting profiles, we remove segments that are partially or fully behind the support profile that is in the back.

These two steps are repeated for each new layer until we've added a predetermined number of profiles.

Algorithm 1: Generate Soma Cube.

Input : S: the size of the Soma cube
Output : T: a 3D array representing the Soma cube
Initialization: $T \leftarrow$ 3D array of size $S \times S \times S$, initialized with zeros; $p = 1$;
Function `GetNeighborhood`(x, y, z, T) **:**
    $neighborhood = \{\}$;
    foreach (x', y', z') *in* $[(x \pm 1, y, z), (x, y \pm 1, z), (x, y, z \pm 1)]$ **do**
        if $0 \leq x', y', z' < S \wedge T[x', y', z'] == 0$ **then**
            $neighborhood = neighborhood \cup \{(x', y', z')\}$;
        end
    end
return $neighborhood$

Function `FillPiece`(i, j, k, p, T) **:**
    $ind \leftarrow \{(i, j, k)\}$;
    for $t \leftarrow 1$ **to** 4 **do**
        if $ind == \emptyset$ **then**
            return;
        end
        $(x, y, z) \leftarrow$ randomly select and remove an element from ind;
        $T[x, y, z] = p$;
        $ind = ind \cup$ `GetNeighborhood`(x, y, z, T) ;
    end
return

while $\exists i, j, k : T[i, j, k] == 0$ **do**
    $(i, j, k) \leftarrow$ randomly select coordinates where $T[i, j, k] == 0$;
    `FillPiece`(i, j, k, p, T) ;
    $p = p + 1$;
end
return T;

4.2 Soma Cubes

The original Soma cube is a 3D spatial reasoning puzzle consisting of seven distinct parts that can be assembled to form a $3 \times 3 \times 3$ cube. It was invented in 1933 by Piet Hein, a Danish polymath, while listening to a lecture on quantum mechanics [40].

An important property of Soma cubes is that all parts can be removed along straight lines. This is a crucial requirement of many approaches that derive blocking relationships between parts, as discussed in Section 2.2.3. In fact, Wan et al. [114] presented an ASP planner that generates sequences optimized for stability, graspability, and accessibility (referred to as assemblability in their paper), and evaluated it using a Soma cube. This planner was later employed by Chen et al. [23] to plan and execute the assembly of a Soma cube using a dual-armed robot. In another work, Kitz and Thomas [67] utilized the planner introduced by Thomas et al. [106] to generate assembly sequences for Soma cubes that were optimized for accessibility.

We implemented a generator that creates Soma cubes of an arbitrary size $S \in \mathbb{N}$. As described in Algorithm 1, the generator iteratively fills an initially empty 3D array T of size $S \times S \times S$ with part identifiers p. The process begins by randomly selecting an empty cell, then growing a part from that cell by setting up to four adjacent empty cells to p. This iteration continues until the entire cube is filled. Finally, each unique part identifier in T is converted to a closed mesh. Examples of generated parts are shown in Figure 4.5, which displays the individual parts of the Soma cube depicted in Figure 4.2.

5 Learning Assembly Actions to Optimize a Single Objective

In Chapter 3, we proposed an efficient approach for planning feasible assembly sequences. However, while having a feasible sequence is a fundamental requirement for assembling any product, it is often not sufficient in practical applications. That is, to be time- and cost-efficient, manufacturers frequently require these sequences to be optimized with respect to objectives such as minimizing assembly time, reducing tool changes, or improving ergonomics for human operators.

We reviewed various approaches to optimal ASP in Section 2.3 that ranged from exhaustive search methods to more sophisticated heuristic-based algorithms. While an exhaustive search is guaranteed to find the optimal solution, it quickly becomes impractical due to the combinatorial explosion of possible sequences. That is, for an assembly with N parts, assuming all sequences are viable, the search space contains $N!$ possible sequences. This limitation has led researchers to explore soft computing techniques such as GA or RL. These methods can efficiently navigate complex search spaces to find near-optimal solutions.

However, all of these optimization approaches start the planning process from scratch for each new assembly. This can be time-consuming and computationally expensive, particularly when manufacturers need to quickly adapt to new or customized product variants.

Building on our approach to feasible ASP, we propose to reuse knowledge from previous planning attempts to accelerate the optimization of assembly sequences. When planning

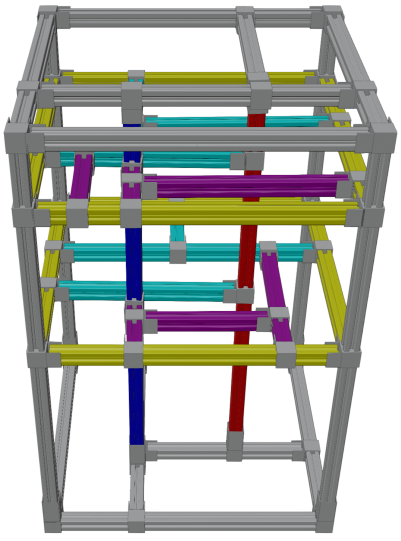


Figure 5.1: Double-layered aluminum profile assembly designed as described in Section 4.1 (reproduced from Cebulla et al. [20]. © 2024, IEEE).

feasible sequences, once a part was identified as removable, it remained removable in all subsequent states. Additionally, the feasibility of the sequence is not affected by when or in which state the part is removed. In contrast, in optimal ASP, the timing of when a part is removed is important, as early removal might significantly improve the overall sequence. For example, removing a specific part early might improve the accessibility of many other parts throughout the remaining assembly process. To capture this temporal aspect and efficiently explore the search space, we utilize an MCTS instead of a DFS. Furthermore, whereas in Chapter 3 we directly used the output of a GNN, here we use it to implement a Q-function.

We evaluated our approach by optimizing the total length of all assembly paths required to assemble aluminum profile assemblies with a robotic manipulator. For that, we used the generator described in Section 4.1 to create two datasets. The first one consists of 14 single-layer assemblies with 21 removable profiles, while the second one contains 7 double-layer assemblies with 30 removable profiles. Examples of such assemblies are depicted by Figures 4.1 and 5.1, respectively. Using leave-one-out cross-validation, our method outperformed a vanilla MCTS by finding assembly sequences with shorter total path lengths, achieving average reductions of 0.8 m for the first dataset and 1.4 m for the second. When training on the first dataset and testing on the second, we found sequences with average total path length reductions of 1.6 m.

In the following sections, we will first position our approach within the related literature. Next, we will formally define the problem of optimizing assembly sequences with respect to a single objective. We then describe how to plan removal paths for individual aluminum profiles. Following this, we discuss our approach to guiding MCTS in efficiently planning assembly sequences. Specifically, we will briefly discuss how ASP can be formalized as an MDP and how we can then use MCTS to search for optimal sequences. We then introduce our graph-based representation of disassembly states and explain how we use DQL with GNNs to guide the MCTS. Finally, we present an evaluation of our approach, demonstrating the effectiveness of our pretrained Q-function in improving MCTS performance.

The work in this chapter has been published in Cebulla et al. [20].

5.1 Positioning Within Related Literature

In Section 2.3.3, we reviewed several methods that reuse knowledge learned from previous planning attempts to plan optimal assembly sequences for new assemblies. We provide an overview in Table 2.2.

The approach proposed by Watanabe and Inada [116] uses tabular Q-learning, which is challenging to scale to large state spaces. Consequently, the authors evaluated their method only on small assemblies containing ten or fewer parts. In contrast, Guo et al. [44] employed DQL, similar to ours and the other discussed approaches. They evaluated their method on larger assemblies with up to 50 parts. However, in contrast to us, they required that the precedence relationships between all parts were known in advance.

Closest to our approach are the works of Hayashi et al. [47] and Funk et al. [39]. Like us, Hayashi et al. [47] used AbD and evaluated their approach on large assemblies. However, as detailed in Section 2.3.4, except for Funk et al. [39], all of these approaches directly

predict the (dis)assembly actions using a learned Q-function. Consequently, incorrect predictions can result in infeasible or suboptimal sequences.

Similar to us, Funk et al. [39] combined MCTS with a learned Q-function. However, in their scenario, a robotic manipulator had to execute each planned action immediately. Consequently, the authors prioritized rapid prediction of assembly actions and limited the search depth of their MCTS to one, i.e., they used it as a single-step look-ahead. Additionally, executed actions could not be undone. As a result, their approach could plan infeasible sequences. This is in stark contrast to the goal of our framework, which is to efficiently plan assembly sequences that are always feasible and approximately optimal with respect to manufacturers' objectives.

5.2 Problem Statement

We extend the problem described in Section 3.2 as follows: let

$$\mathcal{U}_f := \{u | f_f(u) = 1; u \in \mathcal{U}\} \tag{5.1}$$

be the set of all feasible disassembly sequences for an assembly $\mathcal{A} := (\mathcal{P}, \mathcal{C})$, where $\mathcal{U}$ is the set of all disassembly sequences and f_f the feasibility function as defined in Equation 3.2.

Our objective is to find a feasible disassembly sequence $u_f \in \mathcal{U}_f$ that minimizes an objective function $f_o(u) : \mathcal{U}_f \to \mathbb{R}$. In our case, we are specifically interested in minimizing the total length of all the paths that a robotic manipulator must follow to disassemble all parts. Thus, we define $f_o(u)$ as the sum of the lengths of the removal paths a robotic manipulator needs to follow based on the disassembly actions in sequence u_f.

We introduce additional definitions to formalize this. Each part $p_i \in \mathcal{P}$ has grasp points $\mathcal{G}_i$, where each $g_{i,j} = \left(\vec{\tau}, \vec{\Gamma}, \omega\right)$ specifies its position $\vec{\tau} \in \mathbb{R}^3$, the gripper's orientation $\vec{\Gamma} \in \mathrm{SO}(3)$ as well as the width between its fingers ω. For part p_i grasped at $g_{i,j}$, the removal path $q_{g_{i,j}} = \left(\vec{H}_0, \vec{H}_1, \ldots, \vec{H}_M\right)$ is a sequence of gripper poses $\vec{H}_k \in \mathrm{SE}(3)$.

The Euclidean distance between consecutive gripper poses is given by

$$d\left(\vec{H}_k, \vec{H}_{k+1}\right) = \|\vec{t}_k - \vec{t}_{k+1}\|_2, \tag{5.2}$$

where $\vec{t}_k$ is the translational component of $\vec{H}_k$. The length of a removal path $q_{g_{i,j}}$ is then

$$l\left(q_{g_{i,j}}\right) = \sum_{k=0}^{M-1} d\left(\vec{H}_k, \vec{H}_{k+1}\right). \tag{5.3}$$

We now define the shortest removal path q_i^s for a part p_i over all grasp points $\forall g_{i,j} \in \mathcal{G}_i$:

$$q_i^s = q_{g_{i,j}} \quad \text{s.t.} \quad \min_{g_{i,j} \in \mathcal{G}_i} l\left(q_{g_{i,j}}\right) \tag{5.4}$$

As we have defined in Section 3.2, each action $a_t = i$ in the disassembly sequence $u \in \mathcal{U}$ corresponds to the removal of a part p_i at timestep t. Here, we extend this notion such that an action $a_t^q = q_i^s$ in a feasible disassembly sequence

$$u = (s_0, a_0^q), (s_1, a_1^q), \ldots, \left(s_{N-1}, a_{N-1}^q\right), \tag{5.5}$$

corresponds to the shortest removal path q_i^s (the planning of these paths will be discussed in the next section) along which a robotic manipulator can successfully disassemble a specific part p_i.

Finally, we can express our objective function as the sum over all removal path lengths:

$$f_o(u) = \sum_{t=0}^{N-1} l\left(a_t^q\right). \tag{5.6}$$

The overall problem is then to find a feasible disassembly sequence $u \in \mathcal{U}_f$ that minimizes this objective:

$$\min_{u_f \in \mathcal{U}_f} f_o(u). \tag{5.7}$$

5.3 Testing Profile Removability

An important detail of our problem statement is that we optimize over the set of all feasible disassembly sequences $\mathcal{U}_f$. We therefore require a feasibility function f_f. Specifically, we need a method to check if a profile can be removed, i.e., it has at least one grasp point for which a collision-free removal path can be computed.

Given a profile $p_i \in \mathcal{P}$, we approach this computation using a hierarchical method:

1. We first check if any profile blocks p_i by moving it 5 cm in the z-direction while testing for collisions.
2. If no collisions are detected, we proceed to test the reachability of all grasp points $g_{i,j} = \left(\vec{\tau}, \vec{\Gamma}, \omega\right) \in \mathcal{G}_i$. For each grasp point, we attempt to compute the inverse kinematics (IK) such that the robotic gripper is oriented according to $\vec{\Gamma}$, with its two fingers' midpoint placed at $\vec{\tau}$, and the width between them ω. We discard any grasp points for which the IK could not be computed or resulted in collision.
3. Finally, for all reachable grasp points $g_{i,j}^r$, we compute removal paths $q_{g_{i,j}^r}$. If any of these paths is collision-free, p_i is removable.

We utilized the cuRobo [102] motion planning library to parallelize removal path planning, which significantly reduces computation time. Additionally, cuRobo automatically optimizes paths for both length and smoothness.

5.4 Guiding MCTS to Efficiently Plan Assembly Sequences

To solve the problem described in the previous section and find feasible assembly sequences that minimize the total sum of disassembly paths, we extend the framework described in Chapter 3. In particular, we use an MCTS instead of the previously used DFS to search through the DG. This is necessary because removing a specific part early in the sequence can influence the removal path lengths of the subsequently removed parts. Therefore, to efficiently find an optimal sequence, it is important to focus the exploration on states that have this influential part removed. MCTS accomplishes this by employing a tree policy that leverages data gathered during the search to guide exploration towards promising states.

However, MCTS needs to train its policy from scratch for each new assembly, which can be time-consuming. Moreover, optimal ASP significantly increases the complexity of the search space compared to feasible ASP. In detail, for feasible ASP, the order of part removal is irrelevant as long as the precedence constraints are satisfied, and, in the worst case, it requires at most $O(N^2)$ removal tests for an assembly with N parts. In contrast, for optimal ASP, as outlined above, the order is important. In the worst case, an optimal ASP framework needs to evaluate $O(N!)$ potential sequences, which makes optimal ASP considerably more challenging to solve efficiently. To accelerate the optimal ASP, we propose reusing knowledge from previous planning attempts to guide the MCTS. Specifically, we propose implementing a Q-function using a GNN that we then train via DQL.

In the following sections, we first formalize the ASP problem as an MDP. We then explain how MCTS can be applied to search for optimal sequences within this framework. Next, we describe our graph-based representation of disassembly states. We detail how we implement DQL with GNNs to learn a Q-function that predicts the expected cumulative reward of actions. Finally, we explain how the learned Q-function guides the MCTS to efficiently plan assembly sequences that minimize the total removal path length.

5.4.1 Formalizing ASP as an MDP

As we have detailed in Section 3.5.1, we can interpret the disassembly sequence $u_f \in \mathcal{U}_f$ as a rollout of an MDP. In this context, the framework must decide which part to disassemble after each step. Because it is our goal to plan sequences that minimize the total removal path length, we define the immediate reward as the negative value of the removal path's length, $l\left(q_{g_{i,j}}\right)$, which is defined in Equation 5.3:

$$R\left(s_t, a_t, s_{t+1}\right) := -l\left(q_{g_{i,j}}\right). \tag{5.8}$$

5.4.2 MCTS for ASP

One technique to solve the MDP discussed in the previous section is MCTS [16]. It iteratively constructs a search tree to maximize the expected cumulative reward defined in Equation 2.2. This will, in turn, minimize the loss function presented in Equation 5.7.

Each tree node n^T stores a specific disassembly state $s \subseteq S$ as well as a count of its visits $N^T(n^T)$. Additionally, nodes typically track the sum of cumulative rewards obtained over their children. However, we instead recorded the maximum overall cumulative rewards Q^T, which corresponds to the minimum total length for all the disassembly sequences that start from the stored disassembly state s. We did this to reduce the influence of outliers, specifically those with long removal paths. As we discussed in Section 5.3, the motion planning pipeline [102] that we use minimizes the length of its planned paths. However, we could still observe some variability in planned path lengths.

MCTS constructs a search tree through repeating the following four steps until it exhausts a predefined search budget:

Selection

During this step, the already explored search tree is navigated. In detail, starting from the root node n_0^T, a tree policy recursively selects child nodes until it reaches an unexpanded leaf node. We implement this policy via a slightly modified Upper Confidence Bounds for Trees (UCT) [70]:

$$UCT(n^T, n'^T) = \hat{Q}^T(n'^T) + c\sqrt{\frac{2 * \ln N^T(n^T)}{N^T(n'^T)}} \tag{5.9}$$

$$n'^T = \underset{n'^T \in \text{Children}(n^T)}{\text{argmax}} UCT(n^T, n'^T), \tag{5.10}$$

that uses the maximum rather than the average expected reward for its computation. Here, n'^T is a child node of n^T and $\hat{Q}^T$ is the maximum reward normalized to be between 0 and 1. As suggested by Vodopivec et al. [113], for an existing node, we used the maximum and minimum Q^T values of its children as local bounds, whereas for new nodes, we used global bounds. Finally, the constant c balances the exploration of less-visited nodes with the exploitation of nodes with high rewards.

Expansion

After a leaf node with unexplored children is reached, MCTS randomly selects one and adds it to the tree. To do so, it simulates the corresponding disassembly action as we outlined in Section 5.3. Specifically, it computes the shortest collision-free removal path q_i^s for the transition from parent to leaf and stores it in the parent node. As we have previously discussed, our motion pipeline [102] exhibits some variability in path length. To reduce this, we repeat the planning process k times and select the overall shortest path.

Simulation

In this step, MCTS estimates the expected cumulative reward for the node added during the expansion step by simulating the disassembly process. The simulation employs a default policy that selects feasible actions randomly until the disassembly process is complete. In contrast to the expansion step, nodes explored during the simulation step are not added to the search tree. To reduce computation time, we do not plan each removal path multiple times to decrease variability. Instead, we search only once for the shortest path and, as we discussed above, rely on our maximum reward tracking to mitigate variability.

Backpropagation

After the simulation step, the results are propagated upward through the tree. Specifically, the estimated reward values Q^T and visit counts N^T for each node visited during the selection and the expansion steps are adjusted.

After exhausting the search budget, MCTS uses its tree policy to select a child node. This concludes one search step of the current MCTS episode. If this node is not a leaf, it continues the current episode. Otherwise, it starts a new episode, beginning again from the root.

5.4.3 Representing a Disassembly State as a Graph

To represent a disassembly state $s \in S$, we utilize a directed graph $G_{\mathcal{A}} = (N_{\mathcal{A}}, E_{\mathcal{A}})$, which is similar to the graph-based representation we described in Section 3.4. Here, $N_{\mathcal{A}}$ and $E_{\mathcal{A}}$ correspond to the set of nodes and the set of edges, respectively. Each node $n_i \in N_{\mathcal{A}}$ represents a part $p_i \in s$.

However, here we use a fully connected, directed graph, i.e., we add an edge for every pair of nodes n_i and n_j. This contrasts with the previous representation in Section 3.4, where edges represent physical connections and only exist between parts that are in proximity (i.e., within a threshold distance ϵ_c).

In our current representation, each node n_i has a node attribute $\vec{\hat{n}}_i$ that stores the eight corner points of the corresponding part p_i. This provides explicit spatial information about the part's dimensions and orientation within the disassembly state s.

Similarly, each edge $e_{i,j} \in E_{\mathcal{A}}$ has an edge attribute $\vec{\hat{e}}_{i,j}$ that stores a unit vector pointing from the midpoint of part p_i to the midpoint of part p_j as well as the distance between these two points.

5.4.4 DQL with GNNs

As we discussed in Section 2.2.4, DQL [86] is an alternative to MCTS for identifying an optimal policy π^* that maximizes the expected reward defined in Equation 2.2. While traditional Q-learning typically relies on a table to implement the Q-function, DQL instead utilizes a deep neural network. To train this network to approximate the optimal Q-function, DQL utilizes experiences in the form of $(s_t, a_t, s_{t+1}, R(s_t, a_t, s_{t+1}))$ tuples. These are collected during previous explorations of the MDP, which in our context correspond to previous ASP attempts.

To implement the Q-function $Q(s_t, a_t)$, we first compute node embedding for the graph representation, which we discussed in Section 5.4.3. For this, we use the GNN architecture introduced in Section 3.5.2. It processes the input graph G^0 through L layers, where each one updates the node and edge attributes while preserving the graph's structure, as previously defined in Equation 3.13:

$$G^{(l+1)} = \Phi^l\left(G^l\right), \forall l \in [0, L-1].$$

Using its output, we can define our Q-function as:

$$Q(s_t, a_t) = \phi_Q \left(\left[\frac{1}{|N_{s_t}^L|} \sum_{n_j^L \in N_{s_t}^L} \vec{n}_j^L, \vec{n}_{a_t}^L \right] \right).$$

Here, the Q-values are computed by an FNN ϕ_Q that takes as input a state-action pair. The state s_t must summarize the current configuration of the assembly. We compute it as the mean of the node embeddings $\vec{n}_j^L$ over all nodes $N_{s_t}^L$ of the graph $G_{s_t}^L$ computed by the GNN. To represent the action a_t, we utilize the embedding $\vec{n}_{a_t}^L$ of the node corresponding to the to-be removed part.

5.4.5 Guiding MCTS with a Learned Q-function

An important property of DQL is that it is an offline algorithm, which enables us to train our Q-function using experiences gathered from various assemblies. Specifically, we collect these experiences during simulation steps of previous MCTS episodes. For a new assembly, we can then replace the default policy used in the simulation process, as described in Section 5.4.2, with this trained Q-function. To balance between exploiting the predictions of the Q-function and exploring the search tree, we implement an ϵ-greedy strategy. During each step, we generate a random value $r \in [0, 1]$. If $r > \epsilon$, the simulation leverages the pretrained Q-function and tries to remove the part with the highest predicted reward. Conversely, if $r \leq \epsilon$, the simulation uses the default policy, which introduces randomness into the part selection process.

5.5 Evaluation

We performed three experiments to investigate:

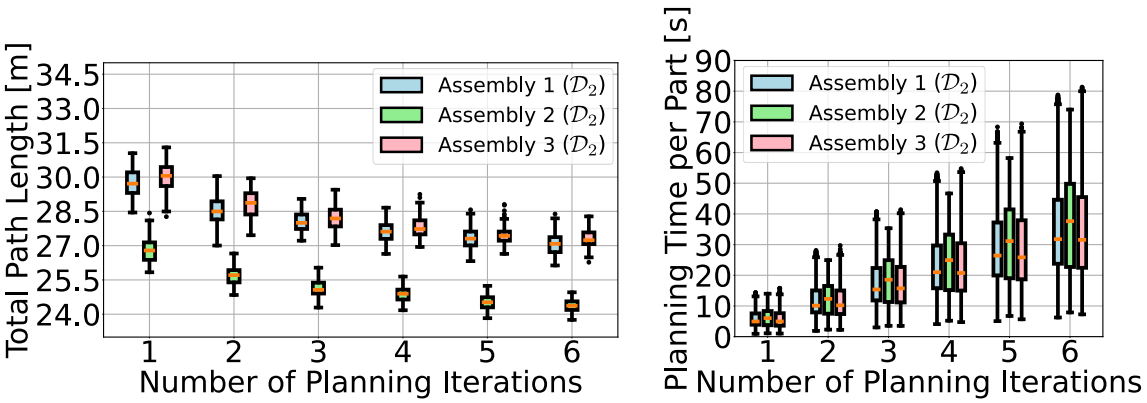
1. The influence of the cuRobo motion planning pipeline [102] on the total removal path length.
2. The effectiveness of a pretrained Q-function in improving MCTS performance.
3. The generalizability of a pretrained Q-function across different levels of assembly complexities.

We used a search budget of five for all MCTS experiments and set $\epsilon = 0.2$ for the ϵ-greedy policy.

In the following sections, we first describe the dataset used for all three experiments, and afterwards we discuss each experiment and its results in detail.

5.5.1 Datasets

Using the generator described in Section 4.1, we created two types of assemblies: 14 single-layer assemblies with 21 removable profiles and 7 double-layer assemblies with 30 removable profiles. To reduce computation time, we kept the bottom profiles and the two



(a) Influence of repeated planning on total path length.

(b) Time required per part for repeated planning.

Figure 5.2: Boxplots showing the total removal path length and planning time required per part versus number of planning iterations (k) for the first three two-layered assemblies over 50 trials. (results published in Cebulla et al. [20]. © 2024, IEEE).

back profiles of the base frame fixed. All assemblies have the same dimensions: 40 cm in width and length, and 60 cm in height. Moreover, all assemblies use two support profiles. As their name suggests, single- and double-layer assemblies have either one or two layers, each with three front profiles. They differ in the number of back profiles, the single-layer assemblies have four back profiles, whereas the double-layer assemblies have three back profiles. We sampled grasp points along each profile at 2 cm intervals for all four sides, while leaving a margin of 3 cm from each end. We collected experiences for both assembly types using the vanilla MCTS described in Section 5.4.2. These experiences were stored in two separate datasets: $\mathcal{D}_1$ for single-layer assemblies and $\mathcal{D}_2$ for double-layer assemblies.

5.5.2 Learning the Q-function

To train our Q-function, we utilized the DQL algorithm discussed in Section 2.2.4. As described in the previous section, the experiences used for training were collected during the selection and simulation steps of previous MCTS episodes and then stored in the corresponding datasets. For each experiment, we then trained the GNN-based Q-function using the respective dataset ($\mathcal{D}_1$ for single-layer assemblies and $\mathcal{D}_2$ for double-layer assemblies). During all experiments, we used a batch size of 128 and trained for 200 episodes.

5.5.3 Analyzing the Removal Path Length Deviation

Our first experiment investigates how the cuRobo [102] motion planning pipeline impacts the total removal path length. Specifically, one stage of this pipeline utilizes a sampling-based planner, which, due to its randomness, can find removal paths of varying lengths for the same part in the same disassembly state. Furthermore, while in a subsequent stage the pipeline optimizes path length, it can only refine previously found paths. In other words, it cannot replace an initially longer path with an entirely different, shorter one.

Table 5.1: Performance comparison of vanilla MCTS versus guided MCTS on single-layer assemblies. For each assembly, we report the total removal path length, number of episodes (#epi), and search steps (#st) required to find the shortest path. The minimum path length for each assembly is highlighted (results published in Cebulla et al. [20]. © 2024, IEEE).

id	Vanilla MCTS for single-layered assemblies			MCTS + Q-function trained on $\mathcal{D}_1$	
	inital path length [m]	shortest path length [m]	found after (#epi, #st)	shortest path length [m]	found after (#epi,#st)
1	16.95	16.53	(15, 1)	**16.07**	(2, 1)
2	15.96	14.55	(14, 7)	**14.37**	(7, 3)
3	16.45	14.73	(2, 6)	**14.63**	(4, 2)
4	16.88	16.16	(3, 1)	**15.08**	(2, 2)
5	14.27	13.64	(20, 10)	**13.34**	(5, 1)
6	18.42	15.51	(9, 1)	**14.12**	(14, 1)
7	16.89	16.65	(8, 3)	**16.03**	(1, 1)
8	18.29	17.22	(3, 3)	**15.99**	(16, 6)
9	17.64	15.48	(7, 1)	**13.87**	(6, 3)
10	19.21	15.52	(16, 7)	**15.37**	(1, 1)
11	18.26	17.65	(7, 3)	**15.12**	(2, 1)
12	17.95	16.35	(18, 3)	**16.16**	(9, 1)
13	17.82	16.70	(5, 3)	**15.78**	(1, 1)
14	17.53	15.23	(15, 2)	**14.69**	(2, 1)

Table 5.2: Performance comparison of vanilla MCTS versus guided MCTS on double-layer assemblies. The Q-function was pretrained either on single-layer assemblies ($\mathcal{D}_1$) or, via leave-one-out cross-validation, on double-layer assemblies ($\mathcal{D}_2$). For each assembly, we report the total removal path length, number of episodes (#epi), and search steps (#st) required to find the shortest path. The minimum path length for each assembly is highlighted (results published in Cebulla et al. [20]. © 2024, IEEE).

id	Vanilla MCTS for double-layered assemblies			MCTS + Q-function trained on $\mathcal{D}_1$		MCTS + Q-function trained on $\mathcal{D}_2$	
	initial path length [m]	shortest path length [m]	found after (#epi,#st)	shortest path length [m]	found after (#epi,#st)	shortest path length [m]	found after (#epi,#st)
1	29.12	24.12	(13, 20)	**23.41**	(6, 1)	24.33	(2, 1)
2	29.31	23.67	(10, 2)	**21.79**	(8, 3)	21.88	(1, 3)
3	30.19	27.70	(13, 1)	**23.88**	(1, 4)	24.03	(13, 4)
4	31.67	27.10	(19, 6)	**24.50**	(2, 2)	25.29	(2, 5)
5	28.82	27.28	(5, 1)	**24.73**	(2, 2)	25.42	(8, 4)
6	26.65	24.31	(17, 4)	24.67	(2, 2)	**23.75**	(7, 7)
7	28.50	26.11	(2, 16)	**25.84**	(1, 3)	26.02	(5, 8)

To analyze the influence of the sampling-based planner on the removal path length and determine if it can be reduced through repeated planning, we conducted the following experiment for each of the first three two-layered assemblies: We identified a feasible robotic disassembly sequence and, for each of its removal paths, performed 1-6 planning iterations, where we only kept the shortest path. For each assembly, we repeated this process 50 times.

The results demonstrate that replanning reduces the influence of random sampling on path length. Figure 5.2a and Figure 5.2b present the total path lengths for each assembly and the corresponding planning time per part, respectively. As can be seen from Figure 5.2a, the median path length decreased by roughly 3 m for all assemblies. However, this comes at the cost of increased planning time, as shown in Figure 5.2b. Specifically, each additional planning iteration adds around 5 s per part. Furthermore, we observed that the reduction in path length diminishes with each additional iteration. As described in Section 5.4.2, we repeat the path planning process during the expansion step of the MCTS. Based on these results, we decided to use four iterations ($k = 4$) for all further experiments.

5.5.4 Evaluating Guided MCTS on the Same Assembly Type

We initially evaluated the pretrained Q-function's effectiveness in improving MCTS performance by separately testing on two datasets: single-layer assemblies ($\mathcal{D}_1$) and double-layer assemblies ($\mathcal{D}_2$). To do so, we utilized a leave-one-out cross-validation strategy for both datasets, where we trained the model on all assemblies except one and then tested it on this remaining assembly.

Single-Layer Assemblies

As can be seen from Table 5.1, our guided MCTS consistently outperformed vanilla MCTS in terms of path length across all single-layer assemblies. Significant improvements, with path length reductions exceeding 0.5 meters, were observed for assemblies 4, 6, 7, 8, 9, 11, 13, and 14. Moreover, the guided MCTS typically found the best path length faster than its vanilla version. Notably, for assemblies 7, 10, and 13, the best path was identified immediately after the initial search step, which suggests that the Q-function alone was sufficient for these particular assemblies.

Double-Layer Assemblies

The results for double-layer assemblies, presented in Table 5.2, support the observations made during the single-layer experiments. That is, utilizing a pretrained Q-function to guide the MCTS simulation process consistently resulted in shorter paths compared to vanilla MCTS. Except for the first and last assemblies, where the guided MCTS produced comparable results to vanilla MCTS, all other double-layer assemblies had path length reductions greater than 0.5 m.

5.5.5 Evaluating Guided MCTS Across Assembly Types

The goal of our final experiment was to evaluate how well a pretrained Q-function generalizes across different levels of assembly complexities. Specifically, we trained a Q-function

on $\mathcal{D}_1$, the single-layer assembly dataset, and evaluated it on $\mathcal{D}_2$, the double-layer assembly dataset. All results are shown in Table 5.2.

Consistent with our intra-dataset experiments, the guided MCTS outperformed vanilla MCTS in both path length optimization and search speed. When guided by the pretrained Q-function, MCTS planned assembly sequences with shorter total removal path lengths for all assemblies except the 6th. In terms of search speed, the guided approach found the best path length within the first two MCTS iterations for all assemblies except the first two.

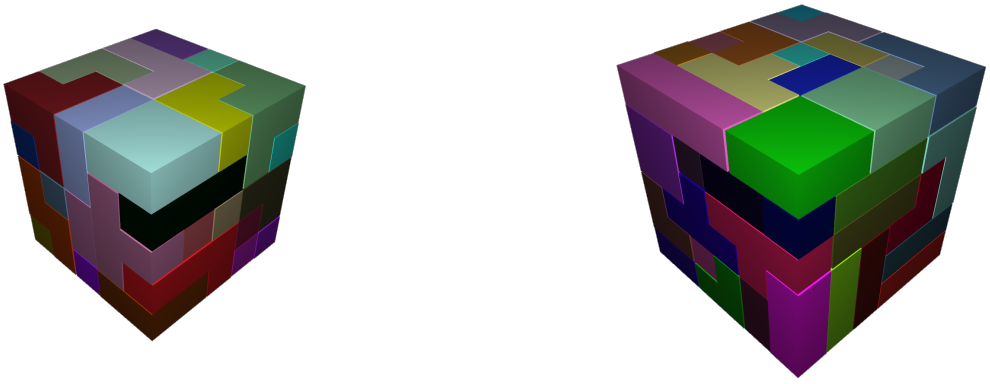
Notably, the Q-function trained on the simpler single-layer dataset ($\mathcal{D}_1$) outperformed the one trained on the target double-layer dataset ($\mathcal{D}_2$) for all assemblies except the 6th. For assemblies 1, 4, and 5, the $\mathcal{D}_1$-trained Q-function found removal paths at least 0.5 m shorter than those found using the $\mathcal{D}_2$-trained Q-function. This suggests that training on simpler assembly tasks may improve generalization in some cases.

6 Learning Assembly Actions to Optimize Multiple Objectives

In Chapter 3, we introduced an approach for efficiently planning feasible assembly sequences. Then, in the previous Chapter 5, we proposed an extension that can efficiently plan assembly sequences optimized for a single objective, such as minimizing the total path length traveled by a robotic manipulator while assembling a product. However, manufacturers often seek to optimize multiple, sometimes conflicting, objectives simultaneously. For example, a manufacturer may want to minimize the number of assembly direction changes to increase efficiency while also maximizing the accessibility of each part throughout the assembly process.

Many of the approaches to optimal ASP that we reviewed in Section 2.3 considered multiple objectives. However, almost all of these methods optimized a linear combination of objectives rather than treating them as separate optimization goals. This approach is straightforward to implement as it can be easily integrated with existing single-objective optimization algorithms. Nevertheless, it has several disadvantages: firstly, manufacturers must predetermine the relative importance of each objective by choosing corresponding weights. If these assumptions about the manufacturing process are incorrect, it can lead to suboptimal solutions. Furthermore, this approach does not provide the necessary insights to identify and fix such issues. In fact, exploring the different trade-offs between objectives requires setting new weights and rerunning the entire optimization. As a result, potentially better solutions might be overlooked.

In contrast, we propose to optimize multiple objectives simultaneously by optimizing their Pareto front. That is, a set of solutions where no objective can be improved without deteriorating at least one other objective. We extend our single-objective ASP opti-



(a) 5×5 Soma cube (38 parts). **(b)** 6×6 Soma cube (58 parts).

Figure 6.1: Two Soma cubes generated with Algorithm 1 (both reproduced from Cebulla et al. [18]. © 2023, IEEE).

mization framework discussed in Chapter 5 such that it can handle Pareto optimization. Furthermore, we explain how to integrate multiple Q-functions into this multi-objective framework to again guide the MCTS and thereby accelerate the ASP process.

We evaluate our approach by optimizing two objectives for Soma cube assemblies: maximizing the accessibility of each part during the assembly process while minimizing the number of direction changes in the sequence. Our experiments utilize two datasets: one containing eight 5×5 Soma cubes and another with four 6×6 Soma cubes. We compare four different settings: vanilla multi-objective MCTS, MCTS with Q-functions trained during the search, MCTS with pretrained Q-functions (our proposed approach), and MCTS with pretrained Q-functions that are retrained during the search. Our results demonstrate that our approach consistently outperforms the other methods, i.e., it found better sequences in less time for both Soma cube sizes.

In the following sections, we will first position our approach within the related literature. Then, we will provide a formal problem definition for multi-objective optimization of assembly sequences. Next, we will describe our method for calculating these objectives using distance maps. Following this, we discuss how we extended our framework to efficiently plan assembly sequences that simultaneously optimize multiple objectives. Specifically, we will explain how multi-objective ASP can be formalized as an MDP with multiple rewards and how we can then use a multi-objective variant of MCTS [91] to search for Pareto-optimal sequences. We then introduce our graph-based representation of disassembly states and explain how we use DQL with GNNs to guide the MCTS. Finally, we present an evaluation of our approach, demonstrating the effectiveness of our pretrained Q-functions in improving MCTS performance for optimizing multiple assembly objectives.

The work in this chapter has been published in Cebulla et al. [18].

6.1 Positioning Within Related Literature

In Section 2.3, we reviewed several approaches to planning optimal assembly sequences for assemblies with and without reusing knowledge from previous attempts. However, among all the reviewed works, only Kiyokawa et al. [68] performed a proper multi-objective, or Pareto, optimization. That is, instead of combining multiple objectives into a single scalar value using weighted sums, they optimized the Pareto front, which represents the set of non-dominated solutions. Each solution in this set is better in at least one objective compared to all others. They did not utilize knowledge transfer in their approach.

Our framework stands out as the only one that combines both knowledge transfer and Pareto optimization for ASP, which offers several advantages over the linear combination approach commonly used in previous works. Specifically, providing a set of Pareto-optimal solutions enables manufacturers to:

- explore trade-offs without making prior assumptions about the relative importance of each objective;
- select a solution based on current priorities or constraints;
- discover solutions that might be missed when using a weighted sum approach.

6.2 Problem Statement

We now extend the single-objective optimal ASP problem presented in Section 5.2 to a multi-objective optimal ASP. To incorporate multiple objectives, we define a vector objective function $\vec{f_o}(u_f) : \mathcal{U}_f \to \mathbb{R}^{m+1}$ as follows:

$$\vec{f_o}(u_f) = (f_{o,0}(u_f), f_{o,1}(u_f), \ldots, f_{o,m}(u_f)), \tag{6.1}$$

where each $f_{o,i}$ for $i \in 0, \ldots, m$ is an individual objective function and $u_f \in \mathcal{U}_f$ represents a feasible disassembly sequence as defined in Equation 5.1.

In particular, we focus on two objectives: $f_{o,0}$ estimates the accessibility of each part during assembly, and $f_{o,1}$ calculates the number of assembly direction changes required to assemble the product. We will provide the implementation details of these objectives in the next section.

Our optimization problem can now be stated, similar to Equation 5.7, as:

$$\max_{u_f \in \mathcal{U}_f} \vec{f_o}(u_f).$$

However, optimizing multiple objectives simultaneously may result in various non-dominated solutions. Therefore, we will optimize the Pareto front. Specifically, given two disassembly sequences $u_f, u'_f \in \mathcal{U}_f$, we say that u_f dominates u'_f if and only if:

$$f_{o,i}(u_f) \geq f_{o,i}(u'_f), \quad \forall i \in \{0, \ldots, m\}, \text{ and} \tag{6.2}$$

$$f_{o,j}(u_f) > f_{o,j}(u'_f), \quad \text{for at least one } j \in \{0, \ldots, m\}. \tag{6.3}$$

That is, u_f is at least as good as u'_f across all objectives and outperforms it in at least one. A solution is considered Pareto optimal if no other solution in the feasible set $\mathcal{U}_f$ dominates it. The Pareto front consists of all Pareto optimal solutions.

6.3 Calculating Objectives

As discussed in Section 2.2.3, an alternative to testing removability via sampling-based motion planners is to precompute the blocking relationships between all parts. These relationships are then stored in specialized data structures for fast removability testing. One example of such a data structure is the $2\frac{1}{2}$ D distance map described by Thomas et al. [106].

In the following, we first briefly discuss the computation of these maps. We then explain how to use them to test for collision-free removal and stability. Finally, we demonstrate how these maps can be utilized to compute geometric accessibility and the number of direction changes.

6.3.1 Distance Maps

One method for computing distance maps was proposed by Thomas et al. [106], who suggested computing the Minkowski difference, as defined in Equation 2.1, between pairs of

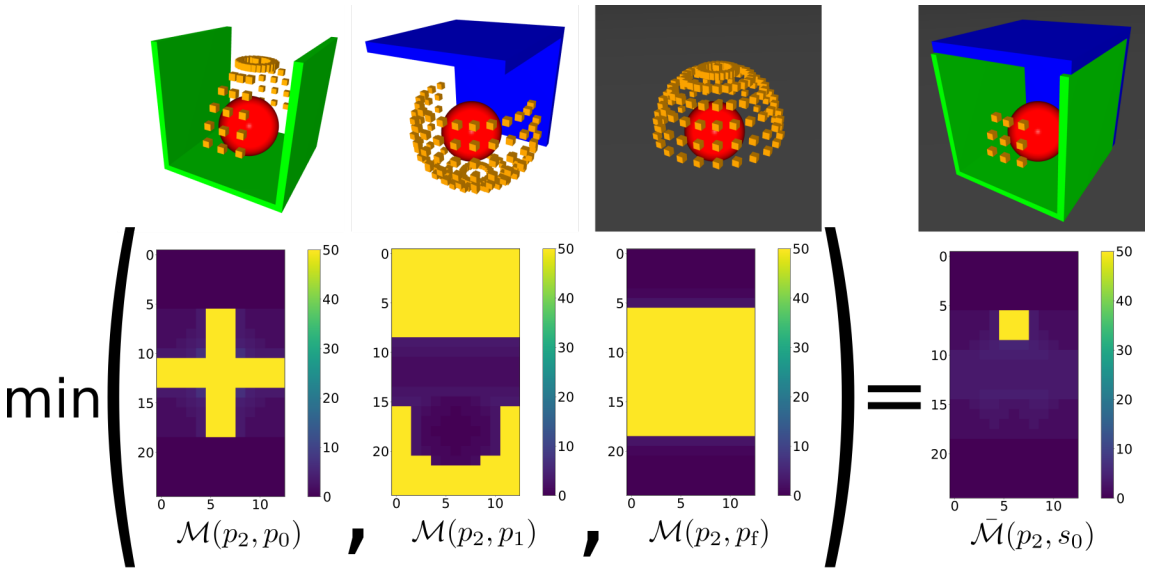


Figure 6.2: Computation of the minimum distance map $\bar{\mathcal{M}}$ for an assembly $\mathcal{A}$ consisting of three parts: green bottom (p_0), blue top (p_1), and red ball (p_2), with the grey floor being a special part (p_f). $\bar{\mathcal{M}}$ is computed for the red ball (p_2) in the initial, assembled state (s_0). Shown at the top are the 3D models of the pairs of parts, as well as the sampled directions (illustrated as orange cubes) along which p_2 can be removed. On the bottom, we depicted their corresponding distance maps $\mathcal{M}$. Yellow entries correspond to directions along which p_2 can be removed.

parts. This results in a set of direction vectors that describe potential collisions when one part is translated by a specific vector. Finally, these vectors are projected via a stereographic projection onto two half-spheres. In contrast, we follow the approach by Andre and Thomas. [8] that samples directions directly from a half sphere along which one part is moved while the other remains static.

Formally, let $\mathcal{M}(p_i, p_j)$ be a $N_M \times N_M$ distance map for parts $p_i \in \mathbf{P}$ and $p_j \in \mathbf{P}$. Each pair of indices (k, l) corresponds to a unique direction vector $\vec{m}$ on the sampled half sphere, while the entry $\mathcal{M}(k, l)$ stores the separation distance d_s. This d_s represents the distance p_i can move along $\vec{m}$ before either a collision with p_j occurs or it reaches a maximum removal distance d_{max}, i.e., it was successfully removed. This is repeated for all pairs of parts, resulting in $|\mathbf{P}|^2 - |\mathbf{P}|$ distance maps.

To determine how far part p_i can be moved relative to all other parts, we compute the minimum over all its corresponding distance maps. Specifically, we define the minimum distance map $\bar{\mathcal{M}}(p_i, s)$ as:

$$\bar{\mathcal{M}}(p_i, s) := \min_{p_j \in s \setminus p_i} \mathcal{M}(p_i, p_j), \tag{6.4}$$

where $s \in S$ is a disassembly state. In other words, $\bar{\mathcal{M}}_{k,l}(p_i, s)$ quantifies the maximum distance part p_i can move without collision in the direction indexed by (k, l) in disassembly state s. Its computation is also illustrated by Figure 6.2.

6.3.2 Collision-Free Removal and Gravity Constraint

Utilizing the minimum distance map $\bar{\mathcal{M}}(p_i, s)$, we can assess if a part p_i can be removed without collision. This is the case if there exists at least one entry in the map, where the separation distance d_s is larger than the maximum removal distance d_{max}. Formally,

$$\exists k, l \in \{0, \ldots, N\} \quad \text{s.t.} \quad \bar{\mathcal{M}}_{k,l}(p_i, s) > d_{\text{max}}. \tag{6.5}$$

To maintain stability in each disassembly state, we must ensure that all remaining parts are supported, which can also be verified with $\bar{\mathcal{M}}$. Namely, after each disassembly action, we check that each remaining part cannot be moved in the direction of gravity (i.e., the minimum distance equals zero), confirming that other parts or the floor fully support it.

6.3.3 Geometric Accessibility Objective

We optimize access to part mounting locations, which offers two advantages. First, providing multiple attachment options increases assembly flexibility, reducing the risk of damaging the part or its neighboring components. Second, better access and greater clearance make the assembly process more robust against imprecision.

We estimated the accessibility of a part p_i in a given disassembly state s as the sum over all the entries of the corresponding minimum distance map $\bar{\mathcal{M}}(p_i, s)$ as defined in Equation 6.4. That is, for a disassembly sequence $u_f \in \mathcal{U}_f$ we can define our first objective, which estimates the accessibility of each part as:

$$f_{o,0}(u_f) = \sum_{t=0}^{N-1} \sum_{k,l \in \{0,\ldots,N_M\}} \bar{\mathcal{M}}_{k,l}(p_{a_t}, s_t). \tag{6.6}$$

6.3.4 Direction Change Objective

Through minimizing the number of direction changes, we reduce the number of movements required to assemble the parts, thereby increasing the overall efficiency and speed of the assembly process. To accomplish this, we focus on maximizing two factors:

1. The number of removal directions for each part.
2. The overlap between the directions available for removing the current part and the removal directions of previously removed parts.

We assume that in the current disassembly state s_n, we want to remove p_i and therefore execute the disassembly action $a_n = i$. We now introduce the notation

$$u_{f,n} = ((s_0, a_0), (s_1, a_1), \ldots, (s_{n-1}, a_{n-1})) \tag{6.7}$$

to denote the feasible disassembly sequence $u_{f,n} \in \mathcal{U}_f$ containing the n previously visited state-action pairs required to reach s_n.

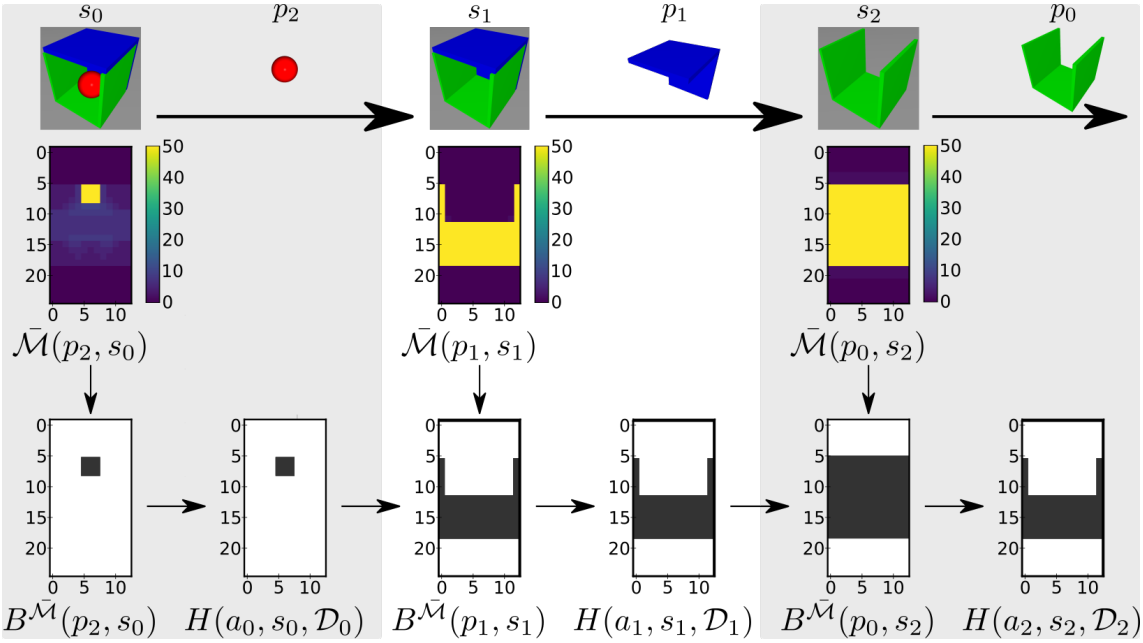


Figure 6.3: Calculation of the direction change objective for a potential assembly sequence of the assembly shown in Figure 6.2.

Additionally, we define $B^{\bar{\mathcal{M}}}(p_i, s)$ as a binary map derived from the minimum distance map $\bar{\mathcal{M}}_{k,l}(p_i, s)$ for part p_i as defined in Equation 6.4. It is computed by:

$$B^{\bar{\mathcal{M}}}_{k,l}(p_i, s) = \begin{cases} 1, & \text{if } \bar{\mathcal{M}}_{k,l}(p_i, s) > d_{\text{max}} \\ 0, & \text{otherwise} \end{cases}. \tag{6.8}$$

That is, $B^{\bar{\mathcal{M}}}_{k,l}(p_i, s) = 1$ indicates that part p_i can be removed along the direction indexed by (k, l) in disassembly state s.

We can now recursively define a function $H(a_n, s_n, u_{f,n})$ that calculates the overlap between the viable removal directions of part p_{a_n} in state s_n as given by $B^{\bar{\mathcal{M}}}(p_{a_n}, s_n)$ and the removal directions of the parts that were previously removed in $u_{f,n}$. Formally,

$$\begin{aligned} H(a_n, s_n, u_{f,n}) = B^{\bar{\mathcal{M}}}(p_{a_n}, s_n) \\ \cap H(a_{i-1}, s_{n-1}, u_{f,n-1}) \end{aligned} \tag{6.9}$$

If there is no overlap between the viable removal directions of the current part p_{a_n} with the previous parts in $u_{f,n}$, we assume that to assemble it, a change in assembly directions is required. Therefore, we restart the computation with all the viable removal directions for part p_{a_n} in state s_n:

$$H(a_n, s_n, u_{f,n}) = B^{\bar{\mathcal{M}}}(p_{a_n}, s_n) \tag{6.10}$$

When performing the first disassembly action a_0 from the full assembly ($s_0 = \mathcal{A}$), we have no prior sequence ($u_{f,0} = ()$). We therefore use all feasible removal directions for part p_{a_0}:

$$H(a_0, s_0, u_{f,0}) = B^{\bar{\mathcal{M}}}(p_{a_0}, s_0).$$

To define our second objective $f_{o,1}$ such that it corresponds to the two factors, we discussed at the start of this section, we first need to introduce a helper function $\hat{H}$ to correctly handle the restart of the computation in case of a direction change as defined in Equation 6.10. If there is no restart, it is equal to the sum of the number of viable removal directions in the current disassembly state that are in common with all previous removal directions:

$$\hat{H}(a_n, s_n, u_{f,n}) = \sum_{k,l \in \{0,\dots,N\}} H_{k,l}(a_n, s_n, u_{f,n}) \tag{6.11}$$

In the case of a direction change, it should evaluate to 0, as we want to minimize these changes. Formally, if $B^{\bar{\mathcal{M}}}(p_{a_n}, s_n) \cap H(a_{n-1}, s_{n-1}, u_{f,n-1}) = \emptyset$:

$$\hat{H}(a_n, s_n, u_{f,n}) = 0 \tag{6.12}$$

Finally, for a disassembly sequence $u_f \in \mathcal{U}_f$ we can define our second objective as:

$$f_{o,1}(u_f) = \sum_{t=0}^{N-1} \hat{H}(a_n, s_n, u_{f,n}). \tag{6.13}$$

Figure 6.3 visualizes these steps for a potential assembly sequence of the assembly shown in Figure 6.2.

6.4 Multi-Objective Optimization for ASP

We utilize a multi-objective variant of MCTS introduced by Perez et al. [91] to search for a Pareto-optimal set of disassembly sequences that minimize assembly direction changes while maximizing the accessibility of each part during its assembly. The computational complexity remains the same as in the single-objective case, i.e., for an assembly with N parts, in the worst case, $N!$ potential disassembly sequences need to be evaluated.

Similar to Chapter 5, we propose reusing knowledge from previous planning attempts to guide the multi-objective MCTS. In detail, because the MCTS optimized multiple objectives, we used separate Q-functions for each one.

In the following sections, we provide a detailed discussion of how we adapted our framework to simultaneously optimize multiple objectives. First, we describe how we formalized the multi-objective ASP problem as an MDP. Next, we explain the multi-objective MCTS [91] variant. Afterwards, we present our approach for guiding this search process using multiple, pretrained Q-functions. Finally, we detail how we represent disassembly states as graphs and then utilize these representations to train GNNs to approximate our Q-functions.

6.4.1 Formalizing Multi-Objective ASP as an MDP

As we extend our framework from single-objective to multi-objective optimization, we also must adapt the MDP formalism, which we introduced in Section 2.2.4. In particular, this requires redefining the reward function. While in Chapter 5 we used a scalar

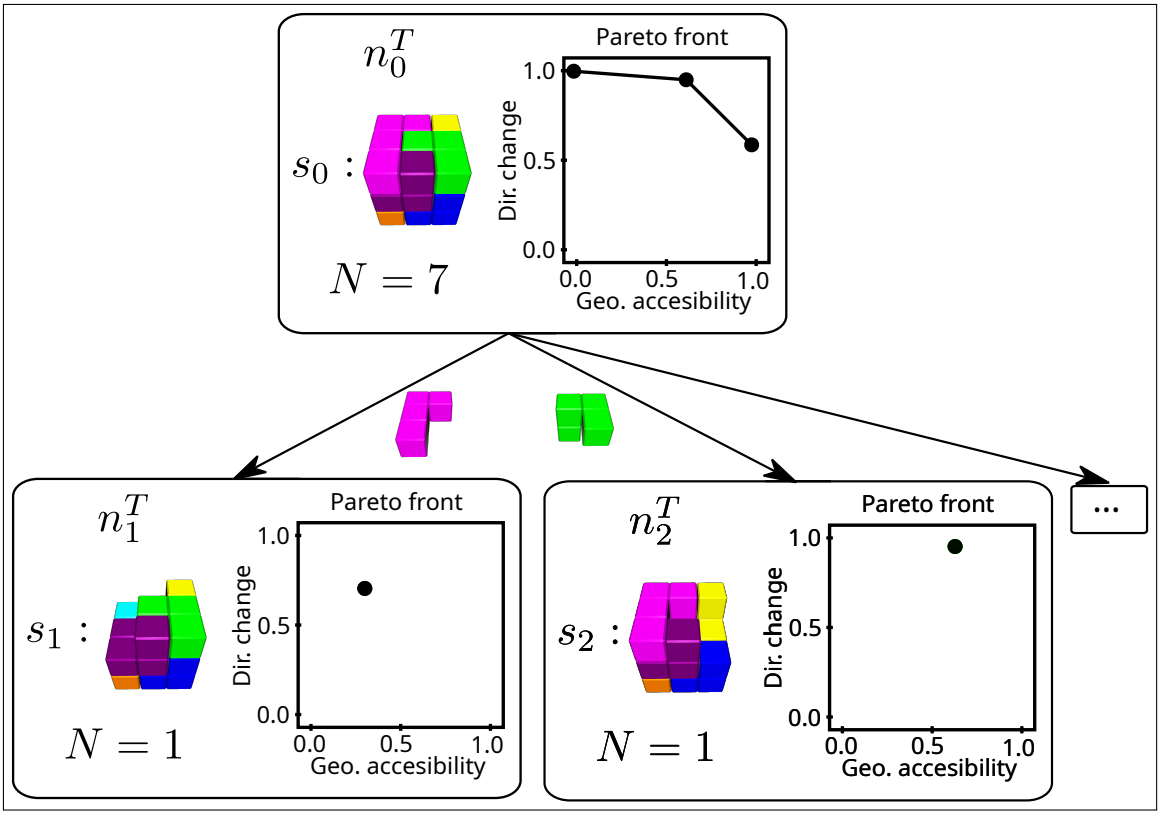


Figure 6.4: A multi-objective variant of MCTS [91] applied to ASP for a Soma cube (reproduced from Cebulla et al. [18]. © 2023, IEEE).

reward function, we now need a vector reward function to express multiple objectives. Specifically, for $m \geq 2$ objectives, we define:

$$\vec{R}(s_t, a_t, s_{t+1}) := (R_1(s_t, a_t, s_{t+1}), R_2(s_t, a_t, s_{t+1}), ..., R_m(s_t, a_t, s_{t+1})). \tag{6.14}$$

Here, $R_i(s_t, a_t, s_{t+1})$ corresponds to reward for the i-th objective, which is gained while transitioning to state s_{t+1} after performing action a in state s_t.

In our case, we have two objectives $f_{o,0}$ and $f_{o,1}$ as defined by Equations 6.6 and 6.13, respectively.

For the first objective, which estimates the accessibility of each part, we define:

$$R_0(s_t, a_t, s_{t+1}) = \sum_{k,l \in \{0,...,N_M\}} \bar{\mathcal{M}}_{k,l}(p_{a_t}, s_t), \tag{6.15}$$

where $\bar{\mathcal{M}}(p_{a_t}, s_t)$ is the minimum distance map defined in Equation 6.4.

For the second objective that maximizes the overlap between all viable disassembly directions, we define:

$$R_1(s_t, a_t, s_{t+1}, u_{f,n}) = \hat{H}(a_n, s_n, u_{f,n}), \tag{6.16}$$

where $\hat{H}(a_n, s_n, u_{f,n})$ is the helper function we defined in Equation 6.11 and 6.12, respectively.

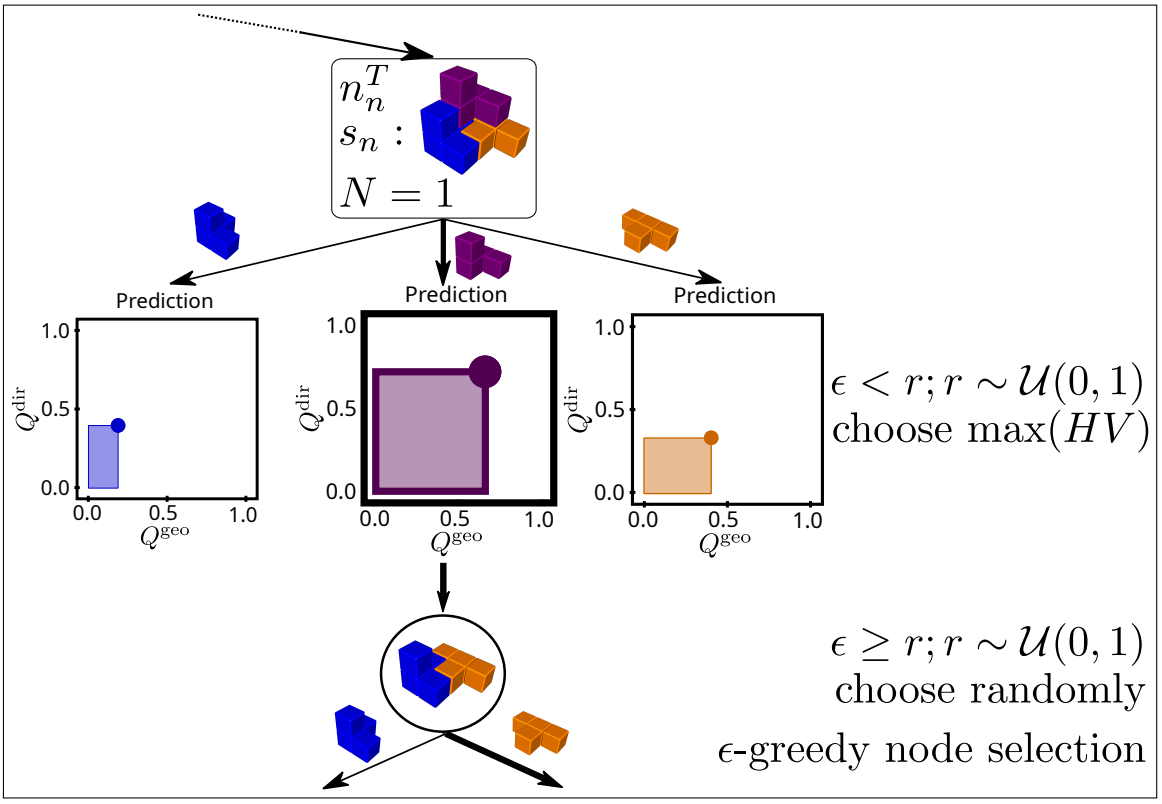


Figure 6.5: A multi-objective MCTS [91] simulation guided by an ϵ-greedy policy. The shaded regions correspond to the hypervolume of the Pareto front (reproduced from Cebulla et al. [18]. © 2023, IEEE).

6.4.2 Multi-Objective MCTS for ASP

To optimize assembly sequences with respect to multiple objectives, we utilize a multi-objective variant of MCTS proposed by Perez et al. [91]. Similar to the single-objective MCTS we described in Section 5.4.2, this algorithm grows a search tree by iteratively performing selection, expansion, simulation, and backpropagation within a given search budget.

In both variants, each tree node n^T stores a disassembly state $s \in S$ as well as N, the number of times it has been visited. Additionally, for the multi-objective variant, each node maintains a local approximation of the Pareto front $\mathcal{P}^f$, which represents the set of non-dominated solutions. Figure 6.4 depicts a search tree with three examples of such nodes.

Furthermore, to handle multiple optimization criteria, the multi-objective MCTS adapts the selection and backpropagation steps. We now provide a detailed explanation of these adaptations:

Selection

Similar to the single-objective MCTS, the selection step utilizes a tree policy to traverse the search tree from the root node n_0^T to an unexpanded leaf node.

In the single-objective case, this policy is commonly implemented based on the UCT [70]:

$$UCT(n^T, n'^T) = \frac{Q^T(n'^T)}{N(n'^T)} + c\sqrt{\frac{2 * \ln N(n^T)}{N(n'^T)}}$$
$$n'^T = \underset{n'^T \in \text{Children}(n^T)}{\text{argmax}} UCT(n^T, n'^T),$$

where n'^T is a child of n^T and Q^T is the total reward obtained for all disassembly sequences that start from the disassembly state s stored in n^T. Further, $N(n^T)$ and $N(n'^T)$ are the visit counts of nodes n^T and n'^T, respectively, and c is a constant balancing exploration and exploitation. In fact, we used a slightly modified version of the UCT described in Equation 5.9 in our single-objective MCTS.

In the multi-objective setting, we need a selection policy that can handle multiple objectives and consider the Pareto front maintained at each node. To achieve this, we adopt the Multi-Objective Upper Confidence Bound (MOUCB) proposed by Perez et al. [91], which utilizes the hypervolume indicator as a measure to evaluate the quality of a Pareto front. In detail, this indicator measures the volume of the objective space delimited by the solutions [128].

Formally, the hypervolume $HV(\mathcal{P}, z)$ of a set of solutions $\mathcal{P}$ relative to a reference point $z \in \mathbb{R}^m$ is defined as:

$$HV(\mathcal{P}, z) = \Lambda\left(\{z' \in \mathbb{R}^m | \exists p \in \mathcal{P} : z \prec z' \prec p\}\right).$$

Here, Λ is the Lebesgue measure, which is a mathematical concept used to determine the "volume" of a subset in $\mathbb{R}^m$. For instance, when $m = 2$, it measures the area, and when $m = 3$, it represents the volume of the space dominated by the solutions in $\mathcal{P}$.

Using this metric, [91] defined the MOUCB as:

$$MOUCB(n^T, n'^T) = \frac{HV(\mathcal{P})}{N(n'^T)} + c\sqrt{\frac{2 * \ln N(n^T)}{N(n'^T)}}.$$

Expansion and Simulation

These two steps are identical to the single-objective MCTS. That is, the expansion step adds an unvisited child node to the tree, while the simulation step estimates the expected reward for this newly added node by simulating the disassembly process until completion.

Backpropagation

The backpropagation step in both variants updates the accumulated rewards in the nodes visited during the selection and expansion steps. In the single-objective MCTS, this involves propagating the estimated cumulative reward Q^T and incrementing the visit counts N^T for each node. The updates are straightforward because the reward is a scalar value representing the total cost.

In the multi-objective MCTS, the backpropagation step is more complex. After the simulation, we obtain a reward vector $\vec{r}$ that contains the cumulative rewards for all objectives. As we backpropagate this vector through the nodes visited during selection and expansion, we test if it either dominates or is dominated by any solution in the local Pareto

fronts $\mathcal{P}^f$ maintained by each node. In detail, at each node, if $\vec{r}$ is dominated by any existing solution in $\mathcal{P}^f$, the vector is discarded, and backpropagation halts. Otherwise, if $\vec{r}$ is not dominated, it is added to $\mathcal{P}^f$, and any solutions in $\mathcal{P}^f$ that are dominated by $\vec{r}$ are removed. This process ensures that each local Pareto front remains consistent and includes only the most promising, non-dominated solutions.

Similar to the single-objective MCTS, these four steps are continuously repeated during each episode until the search budget is depleted. Then, the MOUCB is used to select a child node from which the search resumes. This process continues until a leaf node is found, which marks the end of the current episode. The next episode begins again from the root node n_0^T.

6.4.3 Representing a Disassembly State as a Graph

We represent each disassembly state $s \in S$ using the same graph structure $G_s = (N_s, E_s)$ we described in Section 5.4.3. In this graph, N_s is the set of nodes where each node $n_i \in N_s$ corresponds to a part p_i in s and has an associated node attribute $\vec{u}_{n_i}$ that captures information about that part. The set of edges E_s consists of directed edges $e_{i,j}$ and $e_{j,i}$, each with edge attributes $\vec{v}_{e_{i,j}}$ and $\vec{v}_{e_{j,i}}$, respectively. Because we use a fully connected graph, they are added for each pair of nodes $n_i, n_j \in N_s$.

We utilize this graph structure to create two different graph-based representations for a given disassembly state s: one for the geometric accessibility reward function and another for the direction change reward function.

Graph Representation for Geometric Accessibility

We construct the graph representation G_s^{geo} for the geometric accessibility reward function as defined in Equation 6.15.

For the directed edges, we use the distance maps as described in Section 6.3.1 as attributes: $\vec{v}_{e_{i,j}} = \mathcal{M}(p_i, p_j)$ and $\vec{v}_{e_{j,i}} = \mathcal{M}(p_j, p_i)$.

For the nodes, we assign each node $n_i \in N_s$ the minimum distance map given by Equation 6.4: $\vec{u}_{n_i} := \bar{\mathcal{M}}(p_i, s)$.

Graph Representation for Direction Change

We construct the graph representation $G^{\text{dir}}(s_n, u_{f,n})$ for the direction change reward function as defined in Equation 6.16. This representation follows a similar structure to the geometric accessibility graph but uses binary distance maps instead of continuous ones.

Parallel to the above formulation, the edge attributes are then $\vec{v}_{e_{i,j}} = B^{\mathcal{M}}(p_i, p_j)$ and $\vec{v}_{e_{j,i}} = B^{\mathcal{M}}(p_j, p_i)$. Here, the binary distance map $B^{\mathcal{M}}(p_i, p_j)$ is 1 for all directions in which part p_i can be removed without colliding with p_j, and is 0 otherwise.

For the nodes, we assign each node $n_i \in N_s$ the binary minimum distance map defined in Equation 6.8: $\vec{u}_{n_i} := B^{\bar{\mathcal{M}}}(p_i, s)$.

Unlike other rewards we discussed so far, such as the path length minimization reward as defined in Equation 5.8 or the part accessibility reward, that depend only on the current

disassembly state s_n, this reward is also influenced by the sequence of prior disassembly states and actions $u_{f,n}$ that led to s_n. In more detail, this reward depends on the common removal directions of previously removed parts $H(a_{n-1}, s_{n-1}, u_{f,n-1})$ as given by Equation 6.9. To express this dependence, we leverage the fact that this is also a binary map and add node n_c to store it as an attribute: $\vec{u}_{n_c} := H(a_{n-1}, s_{n-1}, u_{f,n-1})$. We then connect n_c to all other nodes with outgoing edges.

A disadvantage of both representations is that they result in fully connected, directed graphs. While this captures all pairwise interactions between parts, it leads to quadratic growth in the number of edges: for a disassembly state with N parts, we have $|E_s| = N^2 - N$ edges. To reduce computational complexity, we limit the number of edges by considering only the N_c closest neighbors for each part p_i based on the distance maps, adding edges only between these nearest neighbors.

6.4.4 Guiding the Multi-Objective MCTS with Learned Q-functions

Based on the graph representations introduced in the previous section and using the process we described in Section 5.4.4, we can train GNNs to approximate the Q-functions for both the geometric accessibility reward function and the direction change reward function.

Specifically, we compute the Q-function for the geometric accessibility reward as

$$Q^{\text{geo}}(s_t, a_t) = \phi_{\text{geo}}\left(\left[n^L_{a_t}, \frac{1}{|N^{L,\text{geo}}_{s_t}|}\sum_{n^L_j \in N^{L,\text{geo}}_{s_t}} n^L_j\right]\right),$$

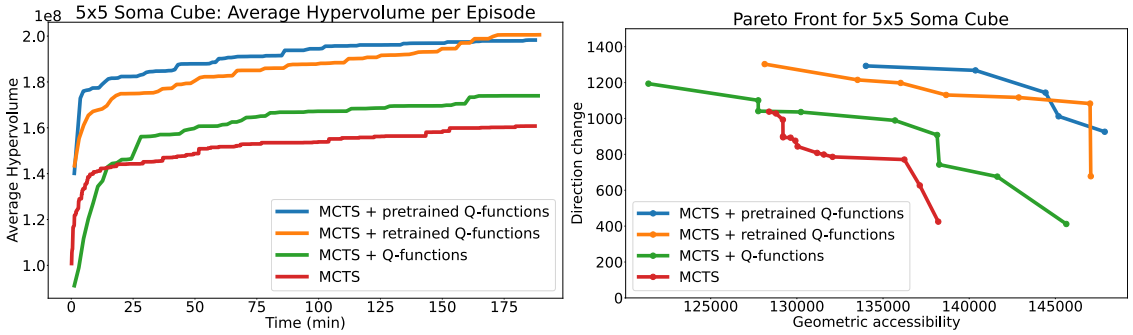
Here, $G^{L,\text{geo}}_s$ is the graph computed for G^{geo}_s by the GNN after L layers. Its corresponding node set is $N^{L,\text{geo}}_s$. We use the node embedding $n^L_{a_t} \in N^{L,\text{geo}}_{s_t}$ to represent action a_t which will remove the corresponding part p_{a_t}. To capture the disassembly state, we compute the global mean over all node embeddings $N^{L,\text{geo}}_{s_t}$. We then concatenate both and provide it as input for an FNN ϕ_{geo}.

For the direction change reward, we define the Q-function Q^{dir} in a similar manner.

We can now use both trained Q-functions to guide the MCTS during the simulation process, as depicted in Figure 6.5. Specifically, we replace the default policy discussed in Section 5.4.2, with an ϵ-greedy policy. That is, during each simulation step, we first randomly choose an $r \sim \mathcal{U}(0, 1)$. If it is larger than a threshold $r > \epsilon$, then we compute the Q values for all feasible actions using the trained Q-functions. We then multiply them for each action, which corresponds to the hypervolume indicator computed for a single solution. Finally, we select the action with the largest product.

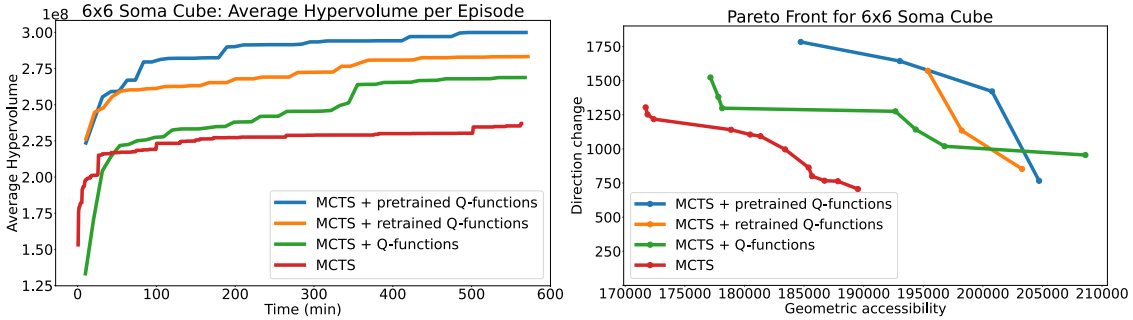
6.5 Evaluation

We evaluated the performance of our multi-objective MCTS guided by learned Q-functions compared to vanilla multi-objective MCTS. Specifically, we consider four settings:



(a) Average hypervolume for 5×5 Soma cubes. **(b)** Pareto front for a 5x5 soma cube.

Figure 6.6: Average hypervolume and Pareto front for 5×5 Soma cubes (results published in Cebulla et al. [18]. © 2023, IEEE).



(a) Average hypervolume for 6×6 Soma cubes. **(b)** Pareto front for a 6x6 soma cube.

Figure 6.7: Average hypervolume and Pareto front for 6×6 Soma cubes (results published in Cebulla et al. [18]. © 2023, IEEE).

1. **MCTS**: Vanilla multi-objective MCTS as baseline.
2. **MCTS + Q-functions**: Q-functions are trained after each episode.
3. **MCTS + pretrained Q-functions**: Our proposed approach using pretrained Q-functions.
4. **MCTS + retrained Q-functions**: Q-functions are pretrained and then retrained after each episode.

All settings used a search budget of 5.

6.5.1 Learning the Q-functions

We performed a series of experiments using two Soma cube datasets. The first dataset consisted of eight 5×5 cubes and the second one consisted of four 6×6 cubes. An example cube for each size is depicted in Figures 6.1a and 6.1b, respectively.

We trained all Q-functions with the DQL [86] algorithm that we described in Section 2.2.4, where we used a replay buffer of size 10000. All experiences gathered during either the selection or the simulation steps were added to this buffer after each MCTS episode. We then sampled batches to train the GNNs using a batch size of five for the 5×5 Soma

cubes and a batch size of one for the 6×6 Soma cubes. Because we wanted to train on 100 samples for both sizes, we ran the training for 20 episodes for the 5×5 Soma cubes and for 100 training episodes for the 6×6 Soma cubes.

6.5.2 Evaluating Multi-Objective Guided MCTS

We present our experimental results for both Soma cube datasets. The results for the 5×5 dataset are shown in Figure 6.6, and those for the 6×6 dataset in Figure 6.7.

We evaluated the search progress by computing the hypervolume of the Pareto front at the root node at each timestep. These values were averaged over all eight cubes for the 5×5 dataset and all four cubes for the 6×6 dataset, as shown in Figures 6.6a and 6.7a. Additionally, for a representative Soma cube from each dataset, we present the final Pareto fronts in Figures 6.6b and 6.7b, respectively.

For both datasets, we required all experiments to run for the same duration. Specifically, for the 6×6 Soma cubes, we used the time needed to run 50 episodes of MCTS, where the Q-functions were trained after each episode **(MCTS + Q-function)**, which was 570 min, as the stopping criterion for all other experiments with this cube size. Similarly, for the 5×5 cubes, we used the time needed to run 100 episodes, which was 190 min.

As can be seen from Figures 6.6a and 6.7a, the **vanilla MCTS** approach underperformed when compared against any of the other methods that utilized Q-functions, despite being computationally faster. On average, it executed 14.4 and 15 times more episodes for the 5×5 and 6×6 cubes, respectively. Our approach, which used pretrained Q-functions **(MCTS + pretrained Q-functions)**, outperformed training Q-functions during the MCTS **(MCTS + Q-function**). Finally, pretrained Q-functions performed better without retraining during the MCTS than with it **(MCTS + retrained Q-functions)**.

7 Discussion, Outlook and Conclusion

Throughout this thesis, we identified and addressed several challenges in automating and optimizing ASP. The goal is to enable efficient planning for customized product variants and to contribute to realizing the vision of Industry 4.0. The work focused on four main areas: improving the efficiency of feasible ASP, optimizing ASP for both single and multiple objectives, and the automatic generation of assemblies from industrial components. We proposed and evaluated novel approaches to all four areas.

In the following sections, we will briefly recapitulate and discuss these approaches. Next, we will provide an overview of the future steps. Finally, we will give some concluding remarks.

7.1 Discussion

The central research questions addressed in this work were how to efficiently

1. identify feasible assembly sequences?
2. generate assemblies with known properties?
3. optimize assembly sequences for a single objective?
4. optimize assembly sequences via Pareto optimization for multiple objectives?

In the following, we summarize the findings and methodologies for each of these research questions.

7.1.1 Efficient Generation of Feasible Assembly Sequences

To address the combinatorial explosion in the search space of feasible assembly sequences, a graph-based representation of assemblies was proposed in Section 3.4. Based on this representation, a GNN was trained to predict part removability, as described in Section 3.5.2. In contrast to related work [125, 41, 83, 10], we utilized these predictions to guide a graph search, efficiently exploring the disassembly graph. That is, we were able to recover if the prediction was wrong. Moreover, we could also handle previously unseen parts. We also planned the assembly sequence in backward (AbD) order. Therefore, we only needed to consider the parts still attached to the assembly at each step. This differs from forward planning approaches, where both assembled and unassembled parts need to be modeled.

The proposed approach significantly reduced the number of unsuccessful removal tests compared to traditional AbD methods. For the evaluated five real-world assemblies, the number of required removal tests was reduced by 64% to 90% compared to the AbD approach proposed by Ebinger et al. [38], and by 30% to 87% when compared to AbD utilizing a bookkeeping heuristic proposed by Dorn et al. [33].

7.1.2 Generation of Assembly Datasets

One of the key challenges in incorporating prior knowledge into the ASP process is the lack of suitable datasets for training and evaluating algorithms. To address this challenge, we developed two assembly generators, which we discussed in Chapter 4.

1. A **generator for 3D aluminum profile assemblies** that a single robotic manipulator can assemble. These assemblies are constructed from real-world industrial parts, enabling us to evaluate the developed framework on practical, complex structures.
2. A **generator for Soma cubes** of arbitrary size. They have the property that each part can be disassembled along a straight line, which is a requirement for most approaches that precompute blocking relations between parts, such as the $2\frac{1}{2}$ D distance maps introduced by Thomas et al. [106] and Andre and Thomas. [8].

7.1.3 Single-Objective Optimization of Assembly Sequences

Building on the feasible ASP framework, a method for efficiently optimizing assembly sequences with respect to a single objective was developed and described in Chapter 5. This approach combines MCTS [16] with DQL [86], using a GNN [12] to learn the Q-function.

The method was evaluated on datasets of aluminum profile assemblies generated using the previously mentioned generator. Specifically, we created two datasets: one consisting of 14 single-layer assemblies with 21 removable profiles, and another containing 7 double-layer assemblies with 30 removable profiles. Our approach consistently outperformed vanilla MCTS in terms of total removal path length. For single-layer assemblies, our method produced sequences with total path lengths that were, on average, 0.8 m shorter. For double-layer assemblies, the improvement increased to an average of 1.4 m shorter paths. Knowledge transfer between datasets of different complexity was also successfully demonstrated, with our method achieving sequences with total path lengths that were, on average, 1.6 m shorter when trained on single-layer assemblies and tested on double-layer assemblies.

7.1.4 Multi-Objective Optimization of Assembly Sequences

When optimizing multiple objectives, all reviewed related work, except for the approach discussed by Kiyokawa et al. [68], optimized a weighted sum of these objectives. This approach is straightforward to implement, but has several disadvantages. First, it requires the prior determination of the relative importance of each objective by assigning corresponding weights. If these assumptions about the manufacturing process are incorrect, it

can lead to suboptimal solutions. Moreover, exploring different trade-offs between objectives requires setting new weights and rerunning the entire optimization. This can be time-consuming and may result in potentially better solutions being overlooked.

To overcome these limitations, we extended our framework to Pareto optimization of assembly sequences, as detailed in Chapter 6. This presented method expanded the single-objective MCTS to handle multiple objectives simultaneously, based on the approach proposed by Perez et al. [91], and incorporated multiple Q-functions to guide the search process. We evaluated this approach on two datasets of Soma cube assemblies created using the generator discussed in Chapter 4. Our goal was to maximize geometric accessibility while minimizing the number of assembly direction changes. During evaluation, our approach outperformed vanilla multi-objective MCTS.

7.2 Outlook

We proposed a framework for efficiently planning feasible assembly sequences that are optimized with respect to various objectives. The main idea was to reuse knowledge gained during previous planning attempts to guide the planning process for new sequences. While we made several contributions to feasible and optimal ASP, as well as in generating suitable datasets, there are various opportunities for future research:

7.2.1 Use of Deep Learning for Feature Extraction

In this work, we utilized three different, hand-crafted features for parts and edges: 3D shape descriptors [62, 9], bounding boxes, and $2\frac{1}{2}$D distance maps [106, 8]. An interesting direction for future research would be to explore deep learning feature extraction techniques. Specifically, we would like to investigate how architectures such as Point Transformer [123], Voxel Networks [126], or PointNet++ [94] could be used to automatically learn such features.

7.2.2 Expanding the Aluminum Profile Generator

Our aluminum profile generator represents a step towards creating more complex assemblies that closely align with industrial needs. However, its current limitation to aluminum profiles arranged in a rectangular shape presents opportunities for expansion in two directions:

1. **Structural Diversity:** The goal is to generate a broader range of shapes beyond the current rectangular configurations. Thus, an expansion should include triangular or hexagonal frames utilizing diagonal profiles. Moreover, it is important to explore the generation of curved or circular assemblies.
2. **Component Variety:** While our current generator focuses on aluminum profiles, real-world industrial settings employ a diverse set of components[1]. To account for

[1] `https://store.boschrexroth.com/Assembly-Technology/Basic-mechanic-elements`

this, incorporating elements such as hinges, wheels, handles, ... is needed. Additionally, common fastening elements, such as screws, nuts, and bolts, require specialized tools for assembly.

Both enhancements would bring our generated assemblies closer to the complexity seen in real-world assemblies, such as those we examined in Section 3.6.1.

7.2.3 Extension to More Complex Assemblies

Alongside the proposed expansion of the aluminum profile generator, the goal is to further extend the developed framework to handle more complex assembly scenarios. Specifically, we intend to remove several assumptions that the current framework makes regarding the properties of assembly sequences, which we discussed in Section 2.1.4:

- **Multiple Hands:** The current framework assumes two-handed, sequential assemblies. To remove this constraint, we need to enable our framework to plan parallel assembly steps using multiple robotic manipulators.
- **Non-Contact-Coherent Assemblies:** Closely linked to the previous point, handling non-contact-coherent assemblies would require the framework to either fix parts, e.g., with clamps, or coordinate actions between multiple agents. For example, one robotic manipulator holds a part in place while another fixes it with a screw.
- **Non-Monotone Assemblies:** The current approach assumes monotonic assembly sequences, where parts, once added, do not need to be moved again. Our framework would need to plan for moving parts that are already attached.
- **Subassemblies:** Currently, the framework focuses on linear assemblies where parts are added one by one. Developing methods to identify and plan for subassemblies that can be constructed separately and then integrated into the main assembly could potentially reduce overall assembly time and complexity.
- **Tool Changes:** Expanding the framework to plan for tool changes is important for handling a wider variety of assembly tasks. We would need to integrate tool selection into the planning process. We could then also optimize sequences to minimize tool changes.

7.3 Conclusion

One of the goals of Industry 4.0 is to shift towards flexible and adaptive manufacturing systems that can efficiently produce customized products. The automation of the ASP process is an important requirement for achieving this goal. One approach to ASP is AbD, which starts with the fully assembled product and iteratively removes parts. After each part is removed, the system performs two checks: first, it tests whether the part could be removed without collisions; second, it verifies that the remaining assembly is stable. This continues until all parts have been removed. Inverting the disassembly sequence then provides an assembly sequence. This is possible if no irreversible joining methods, such as welding or gluing, are used. However, these methods face a combinatorial explosion

in the search space. For an assembly with N parts, finding a feasible assembly sequence requires $O(N^2)$ tests in the worst case. Moreover, the size of the search space further increases if not only any feasible sequence is required, but one optimized with respect to one or more objectives. In this case, the search space can grow to $N!$ potential sequences, as the order in which parts are removed becomes crucial for optimization.

The focus of this work was to address this challenge by accelerating the ASP process through the reuse of knowledge from previous planning attempts. To this end, we made the following contributions:

Efficient Identification of Removable Parts

We developed a learning-based approach for identifying assembly parts that have a high likelihood of being removable. It utilizes a graph-based assembly representation and a GNN to learn from previous planning attempts. The predictions of the GNN are then utilized by a graph search to test the most promising parts for removal first. With this approach, we could significantly reduce the number of unsuccessful removal tests, thus improving the overall efficiency of the ASP process.

Efficient Discovery of Assembly Sequence Optimized for a Single Objectives

We proposed a strategy for efficiently discovering optimal assembly sequences by combining MCTS with DQL. Specifically, we used a pretrained Q-function, implemented using a GNN, to guide the simulation step of an MCTS. During this step, we employed an ϵ-greedy policy. With probability $1 - \epsilon$, the policy selects the action with the highest Q-value to exploit the knowledge captured by the Q-function. With probability ϵ, it selects a random action to encourage exploration of potentially promising actions. Our approach enables more efficient exploration of the complex search space of optimal assembly sequences, significantly reducing the time required to find high-quality solutions.

Efficient Discovery of Assembly Sequence Optimized for Multiple Objectives

We extended the framework to simultaneously optimize multiple, potentially conflicting objectives. To do so, we utilize a variant of MCTS that stores a local Pareto front at each node. We then trained separate Q-functions for each objective and used them to guide the MCTS. Specifically, during the simulation step of MCTS, we again employed an ϵ-greedy policy as described earlier, with the exploitation step selecting actions based on the product of Q-values across all objectives. Our experiments demonstrated that this guided multi-objective MCTS consistently outperformed the vanilla approach, i.e., it produced Pareto fronts with larger hypervolumes in less computation time.

Assembly Generators for Training and Evaluating ASP Algorithms

To address the lack of high-quality datasets for training and evaluating ASP methods, we developed two assembly generators. The first generator creates 3D aluminum profile assemblies that a single robot manipulator can assemble. The second generator creates Soma cubes, which are a type of 3D puzzle. These have the property that all parts can be removed along straight trajectories, which is a requirement for various ASP algorithms that we reviewed in Section 2.2, which precompute blocking relationships between parts.

Bibliography

[1] Premium panel cart. URL https://grabcad.com/library/premium-panel-cart-1.

[2] Angled cart/lectern with drawer. URL https://grabcad.com/library/angled-cart-lectern-with-drawer-1.

[3] Aluminium profile shelf. URL https://grabcad.com/library/aluminium-profile-shelf-1.

[4] Large rotatable white board. URL https://grabcad.com/library/large-rotatable-white-board-1.

[5] Kjn workbench 4. URL https://grabcad.com/library/kjn-workbench-4-1.

[6] Eren Erdal Aksoy, Alexey Abramov, Johannes Dörr, Kejun Ning, Babette Dellen, and Florentin Wörgötter. Learning the semantics of object–action relations by observation. *The International Journal of Robotics Research*, 30(10):1229–1249, 2011.

[7] Robert Andre and Ulrike Thomas. Anytime assembly sequence planning. In *Proceedings of ISR 2016: 47st International Symposium on Robotics*, pages 1–8. VDE, 2016.

[8] Robert Andre and Ulrike Thomas. Error robust and efficient assembly sequence planning with haptic rendering models for rigid and non-rigid assemblies. In *2017 IEEE International Conference on Robotics and Automation (ICRA)*, 2017. doi: 10.1109/ICRA.2017.8262698.

[9] Mihael Ankerst, Gabi Kastenmüller, Hans-Peter Kriegel, and Thomas Seidl. 3d shape histograms for similarity search and classification in spatial databases. In *Advances in Spatial Databases: 6th International Symposium, SSD'99 Hong Kong, China, July 20—23, 1999 Proceedings 6*, pages 207–226. Springer, 1999.

[10] Matan Atad, Jianxiang Feng, Ismael Rodríguez, Maximilian Durner, and Rudolph Triebel. Efficient and feasible robotic assembly sequence planning via graph representation learning. In *2023 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 8262–8269, 2023. doi: 10.1109/IROS55552.2023.10342352.

[11] MVA Raju Bahubalendruni and Bibhuti Bhusan Biswal. An intelligent approach towards optimal assembly sequence generation. *Proceedings of the Institution of Mechanical Engineers, Part C: Journal of Mechanical Engineering Science*, 232 (4):531–541, 2018.

[12] Peter Battaglia, Jessica Blake Chandler Hamrick, Victor Bapst, Alvaro Sanchez, Vinicius Zambaldi, Mateusz Malinowski, Andrea Tacchetti, David Raposo, Adam Santoro, Ryan Faulkner, Caglar Gulcehre, Francis Song, Andy Ballard, Justin Gilmer, George E. Dahl, Ashish Vaswani, Kelsey Allen, Charles Nash, Victoria Jayne Langston, Chris Dyer, Nicolas Heess, Daan Wierstra, Pushmeet Kohli, Matt Botvinick, Oriol Vinyals, Yujia Li, and Razvan Pascanu. Relational inductive biases, deep learning, and graph networks. *ArXiv*, abs/1806.01261, 2018.

[13] Dmitriy Bespalov, Cheuk Yiu Ip, William C. Regli, and Joshua Shaffer. Benchmarking cad search techniques. In *Symposium on Solid and Physical Modeling*, 2005.

[14] Aude G. Billard, Sylvain Calinon, and Rüdiger Dillmann. Learning from humans. *Springer handbook of robotics*, pages 1995–2014, 2016.

[15] Alain Bourjault. Contribution à une approche méthodologique de l'assemblage automatisé: élaboration automatique des séquences opératoires. *These D'etat, Universite de Franche-Comte*, 1984.

[16] Cameron Browne, Edward Jack Powley, Daniel Whitehouse, Simon M. M. Lucas, Peter I. Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez Liebana, Spyridon Samothrakis, and Simon Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4: 1–43, 2012.

[17] Berk Calli, Aaron Walsman, Arjun Singh, Siddhartha Srinivasa, Pieter Abbeel, and Aaron M Dollar. Benchmarking in manipulation research: Using the yale-cmu-berkeley object and model set. *IEEE Robotics & Automation Magazine*, 22(3): 36–52, 2015.

[18] Alexander Cebulla, Tamim Asfour, and Torsten Kröger. Efficient multi-objective assembly sequence planning via knowledge transfer between similar assemblies. In *2023 IEEE 19th International Conference on Automation Science and Engineering (CASE)*, pages 1–7. IEEE, 2023.

[19] Alexander Cebulla, Tamim Asfour, and Torsten Kröger. Speeding up assembly sequence planning through learning removability probabilities. In *2023 IEEE International Conference on Robotics and Automation (ICRA)*, pages 12388–12394. IEEE, 2023.

[20] Alexander Cebulla, Tamim Asfour, and Torsten Kröger. Beyond feasibility: Efficiently planning robotic assembly sequences that minimize assembly path lengths. In *2024 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Abu Dhabi, UAE, October 2024. IEEE. To appear.

[21] Sugato Chakrabarty and Jan Wolter. A hierarchical approach to assembly planning. In *Proceedings of the 1994 IEEE International Conference on Robotics and Automation*, pages 258–263 vol.1, 1994. doi: 10.1109/ROBOT.1994.350979.

[22] Chih-Chung Chang and Chih-Jen Lin. Libsvm: A library for support vector machines. *ACM Trans. Intell. Syst. Technol.*, 2:27:1–27:27, 2011.

[23] Hao Chen, Weiwei Wan, Keisuke Koyama, and Kensuke Harada. Planning to build block structures with unstable intermediate states using two manipulators. *IEEE Transactions on Automation Science and Engineering*, 19(4):3777–3793, 2022. doi: 10.1109/TASE.2021.3136006.

[24] Jack Collins, Mark Robson, Jun Yamada, Mohan Sridharan, Karol Janik, and Ingmar Posner. Ramp: A benchmark for evaluating robotic assembly manipulation and planning. *IEEE Robotics and Automation Letters*, 2023.

[25] K Collins, AJ Palmer, and K Rathmill. The development of a european benchmark for the comparison of assembly robot programming systems. In *Robot Technology and Applications: Proceedings of the 1st Robotics Europe Conference Brussels, June 27–28, 1984*, pages 187–199. Springer, 1985.

[26] Juan Cortés, Léonard Jaillet, and Thierry Siméon. Disassembly path planning for complex articulated objects. *IEEE Transactions on Robotics*, 24(2):475–481, 2008.

[27] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and Thirunavukarasu Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, 2002. doi: 10.1109/4235.996017.

[28] BBVL Deepak, G Bala Murali, MVA Raju Bahubalendruni, and BB Biswal. Assembly sequence planning using soft computing methods: a review. *Proceedings of the Institution of Mechanical Engineers, Part E: Journal of Process Mechanical Engineering*, 233(3):653–683, 2019.

[29] Thomas L. DeFazio and Daniel E. Whitney. Simplified generation of all mechanical assembly sequences. *IEEE Journal on Robotics and Automation*, 3(6):640–658, 1987. doi: 10.1109/JRA.1987.1087132.

[30] Gino Dini and Marco Santochi. Automated sequencing and subassembly detection in assembly planning. *CIRP annals*, 41(1):1–4, 1992.

[31] Sebastian Dorn, Nicola Wolpert, and Elmar Schömer. Voxel-based general voronoi diagram for complex data with application on motion planning. *2020 IEEE International Conference on Robotics and Automation (ICRA)*, pages 137–143, 2020.

[32] Sebastian Dorn, Nicola Wolpert, and Elmar Schömer. Expansive voronoi tree: A motion planner for assembly sequence planning. *2021 IEEE International Conference on Robotics and Automation (ICRA)*, pages 7880–7886, 2021.

[33] Sebastian Dorn, Nicola Wolpert, and Elmar Schomer. An assembly sequence planning framework for complex data using general voronoi diagram. *2022 International Conference on Robotics and Automation (ICRA)*, pages 9896–9902, 2022.

[34] Christian R. G. Dreher and Tamim Asfour. Learning temporal task models from human bimanual demonstrations. In *2022 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 7664–7671. IEEE, 2022.

[35] Christian R. G. Dreher, Mirko Wächter, and Tamim Asfour. Learning object-action relations from bimanual human demonstration using graph networks. *IEEE Robotics and Automation Letters*, 5(1):187–194, 2019.

[36] Christian R. G. Dreher, Manuel Zaremski, Fabian Leven, David Schneider, Alina Roitberg, Rainer Stiefelhagen, Michael Heizmann, Barbara Deml, and Tamim Asfour. Erfassung und interpretation menschlicher handlungen für die programmierung von robotern in der produktion. *at-Automatisierungstechnik*, 70(6):517–533, 2022.

[37] Vijay Prakash Dwivedi and Xavier Bresson. A generalization of transformer networks to graphs. *AAAI Workshop on Deep Learning on Graphs: Methods and Applications*, 2021.

[38] Timothy Ebinger, Sascha Kaden, Shawna L. Thomas, Robert Andre, Nancy M. Amato, and Ulrike Thomas. A general and flexible search framework for disassembly planning. *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1–8, 2018.

[39] Niklas Funk, Georgia Chalvatzaki, Boris Belousov, and Jan Peters. Learn2assemble with structured representations and search for robotic architectural construction. In *Conference on Robot Learning*, pages 1401–1411. PMLR, 2022.

[40] Martin Gardner. *The 2nd Scientific American Book of Mathematical Puzzles & Diversions*. Simon & Schuster, New York, 1961. ISBN 0-226-28253-8. Reprinted in 1987 by University of Chicago Press.

[41] Seyed Kamyar Seyed Ghasemipour, Satoshi Kataoka, Byron David, Daniel Freeman, Shixiang Shane Gu, and Igor Mordatch. Blocks assemble! learning to assemble with large-scale structured reinforcement learning. In *International Conference on Machine Learning*, pages 7435–7469. PMLR, 2022.

[42] David E. Goldberg and Robert Lingle. Alleles, loci, and the traveling salesman problem. In John J. Grefenstette, editor, *Proceedings of an International Conference on Genetic Algorithms and Their Applications*, pages 154–159, Hillsdale, New Jersey, 1985. Lawrence Erlbaum Associates.

[43] Leonidas J. Guibas, Dan Halperin, Hirohisa Hirukawa, Jean-Claude Latombe, and Randall H. Wilson. A simple and efficient procedure for polyhedral assembly partitioning under infinitesimal motions. *Proceedings of 1995 IEEE International Conference on Robotics and Automation*, 3:2553–2560 vol.3, 1995.

[44] Kai Guo, Rui Liu, Guijiang Duan, Jiajun Liu, and Pengyong Cao. Research on dynamic decision-making for product assembly sequence based on connector-linked model and deep reinforcement learning. *Journal of Manufacturing Systems*, 71: 451–473, 2023.

[45] Dan Halperin, Jean-Claude Latombe, and Randall H. Wilson. A general framework for assembly planning: The motion space approach. *Algorithmica*, 26:577–601, 2000.

[46] Nikolaus Hansen and Andreas Ostermeier. Adapting arbitrary normal mutation distributions in evolution strategies: the covariance matrix adaptation. In *Proceedings of IEEE International Conference on Evolutionary Computation*, pages 312–317, 1996. doi: 10.1109/ICEC.1996.542381.

[47] Kazuki Hayashi, Makoto Ohsaki, and Masaya Kotera. Assembly sequence optimization of spatial trusses using graph embedding and reinforcement learning. *Journal of the International Association for Shell and Spatial Structures*, 63(4): 232–240, 2022.

[48] Robert Hegewald, Nicola Wolpert, and Elmar Schomer. Iterative mesh modification planning: A new method for automatic disassembly planning of complex industrial components. *2022 International Conference on Robotics and Automation (ICRA)*, pages 336–342, 2022.

[49] Robert Hegewald, Nicola Wolpert, and Elmar Schömer. Iterative mesh modification planning: A new method for automatic disassembly planning of complex industrial components. In *2022 International Conference on Robotics and Automation (ICRA)*, pages 336–342, 2022. doi: 10.1109/ICRA46639.2022.9812116.

[50] Matteo Hessel, Joseph Modayil, Hado Van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Azar, and David Silver. Rainbow: Combining improvements in deep reinforcement learning. In *Proceedings of the AAAI conference on artificial intelligence*, volume 32, 2018.

[51] Luiz Scaramelli Homem de Mello and Arthur C. Sanderson. And/or graph representation of assembly plans. *IEEE Trans. Robotics Autom.*, 6:188–199, 1986.

[52] Luiz Scaramelli Homem de Mello and Arthur C. Sanderson. A correct and complete algorithm for the generation of mechanical assembly sequences. In *Proceedings, 1989 International Conference on Robotics and Automation*, pages 56–61 vol.1, 1989. doi: 10.1109/ROBOT.1989.99967.

[53] Luiz Scaramelli Homem de Mello and Arthur C. Sanderson. Representations of mechanical assembly sequences. *IEEE Transactions on Robotics and Automation*, 7(2):211–227, 1991. doi: 10.1109/70.75904.

[54] David Hsu, Jean-Claude Latombe, and Rajeev Motwani. Path planning in expansive configuration spaces. *Proceedings of International Conference on Robotics and Automation*, 3:2719–2726 vol.3, 1997.

[55] David Hsu, Lydia E. Kavraki, Jean-Claude Latombe, Rajeev Motwani, and Stephen Sorkin. On finding narrow passages with probabilistic roadmap planners. In *Robotics: The Algorithmic Perspective: 1998 Workshop on the Algorithmic Foundations of Robotics*, pages 141–154, March 1998.

[56] Auke Jan Ijspeert, Jun Nakanishi, Heiko Hoffmann, Peter Pastor, and Stefan Schaal. Dynamical movement primitives: learning attractor models for motor behaviors. *Neural computation*, 25(2):328–373, 2013.

[57] Natraj Iyer, Subramaniam Jayanti, Kuiyang Lou, Yagnanarayanan Kalyanaraman, and Karthik Ramani. Three-dimensional shape searching: state-of-the-art review and future trends. *Computer-Aided Design*, 37(5):509–530, 2005.

[58] Pablo Jiménez. Survey on assembly sequencing: a combinatorial and geometrical perspective. *Journal of Intelligent Manufacturing*, 24:235–250, 2013.

[59] Rondall E. Jones, Randall H. Wilson, and Terri L. Calton. On constraints in assembly planning. *IEEE Trans. Robotics Autom.*, 14:849–863, 1998.

[60] Rondall E. Jones, Randall H. Wilson, and Terri L. Calton. On constraints in assembly planning. *IEEE Transactions on Robotics and Automation*, 14(6):849–863, 1998.

[61] Lydia E. Kavraki, Petr Svestka, Jean-Claude Latombe, and Mark H. Overmars. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *IEEE Trans. Robotics Autom.*, 12:566–580, 1996.

[62] Michael Kazhdan, Thomas Funkhouser, and Szymon Rusinkiewicz. Rotation invariant spherical harmonic representation of 3d shape descriptors. In *Symposium on geometry processing*, volume 6, pages 156–164, 2003.

[63] Maurice G. Kendall. A new measure of rank correlation. *Biometrika*, 30(1-2): 81–93, 1938.

[64] Sangpil Kim, Hyung-gun Chi, Xiao Hu, Qixing Huang, and Karthik Ramani. A large-scale annotated mechanical components benchmark for classification and retrieval tasks with deep neural networks. In *Computer Vision–ECCV 2020: 16th European Conference, Glasgow, UK, August 23–28, 2020, Proceedings, Part XVIII 16*, pages 175–191. Springer, 2020.

[65] Kenneth Kimble, Karl Van Wyk, Joe Falco, Elena Messina, Yu Sun, Mizuho Shibata, Wataru Uemura, and Yasuyoshi Yokokohji. Benchmarking protocols for evaluating small parts robotic assembly systems. *IEEE robotics and automation letters*, 5(2):883–889, 2020.

[66] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.

[67] Kristof Kitz and Ulrike Thomas. Neural dynamic assembly sequence planning. In *2021 IEEE 17th International Conference on Automation Science and Engineering (CASE)*, pages 2063–2068, 2021. doi: 10.1109/CASE49439.2021.9551620.

[68] Takuya Kiyokawa, Jun Takamatsu, and Tsukasa Ogasawara. Assembly sequences based on multiple criteria against products with deformable parts. In *2021 IEEE International Conference on Robotics and Automation (ICRA)*, pages 975–981. IEEE, 2021.

[69] Sebastian Koch, Albert Matveev, Zhongshi Jiang, Francis Williams, Alexey Artemov, Evgeny Burnaev, Marc Alexa, Denis Zorin, and Daniele Panozzo. Abc: A big cad model dataset for geometric deep learning. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 9601–9611, 2019.

[70] Levente Kocsis and Csaba Szepesvari. Bandit based monte-carlo planning. In *European Conference on Machine Learning*, 2006.

[71] Nathan P. Koenig and Andrew Howard. Design and use paradigms for gazebo, an open-source multi-robot simulator. In *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, volume 3, pages 2149–2154 vol.3, 2004. doi: 10.1109/IROS.2004.1389727.

[72] Janet L. Kolodner. An introduction to case-based reasoning. *Artificial intelligence review*, 6(1):3–34, 1992.

[73] Oliver Kroemer, Scott Niekum, and George Dimitri Konidaris. A review of robot learning for manipulation: Challenges, representations, and algorithms. *J. Mach. Learn. Res.*, 22:30:1–30:82, 2019.

[74] Hsin-Yi Lai and Chin-Tzwu Huang. A systematic approach for automatic assembly sequence plan generation. *The International Journal of Advanced Manufacturing Technology*, 24:752–763, 2004.

[75] Steven LaValle. Rapidly-exploring random trees: A new tool for path planning. *Research Report 9811*, 1998.

[76] Beatrice Lazzerini, Francesco Marcelloni, Gino Dini, and Franco Failli. Assembly planning based on genetic algorithms. In *18th International Conference of the North American Fuzzy Information Processing Society-NAFIPS (Cat. No. 99TH8397)*, pages 482–486. IEEE, 1999.

[77] Duc Thanh Le, Juan Cortés, and Thierry Siméon. A path planning approach to (dis)assembly sequencing. *2009 IEEE International Conference on Automation Science and Engineering*, pages 286–291, 2009.

[78] Kimoon Lee, Sungmoon Joo, and Henrik I. Christensen. An assembly sequence generation of a product family for robot programming. In *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 1268–1274, 2016. doi: 10.1109/IROS.2016.7759210.

[79] Youngwoon Lee, Edward S. Hu, and Joseph J. Lim. Ikea furniture assembly environment for long-horizon complex manipulation tasks. In *2021 ieee international conference on robotics and automation (icra)*, pages 6343–6349. IEEE, 2021.

[80] Guohao Li, Chenxin Xiong, Ali K. Thabet, and Bernard Ghanem. Deepergcn: All you need to train deeper gcns. *ArXiv*, abs/2006.07739, 2020.

[81] Tao Lu, Bo Zhang, and Peifa Jia. Assembly sequence planning based on graph reduction. *Proceedings of TENCON '93. IEEE Region 10 International Conference on Computers, Communications and Automation*, 4:119–122 vol.4, 1993.

[82] Katia Lupinetti, Franca Giannini, Marina Monti, and Jean-Philippe Pernot. Content-based multi-criteria similarity assessment of cad assembly models. *Computers in Industry*, 112:103111, 2019.

[83] Lin Ma, Jiangtao Gong, Hao Xu, Hao Chen, Hao Zhao, Wenbing Huang, and Guyue Zhou. Planning assembly sequence with graph transformer. In *2023 IEEE International Conference on Robotics and Automation (ICRA)*, pages 12395–12401, 2023. doi: 10.1109/ICRA48891.2023.10160424.

[84] Romeo M. Marian, Lee H. S. Luong, and Kazem Abhary. A genetic algorithm for the optimisation of assembly sequences. *Computers & Industrial Engineering*, 50 (4):503–527, 2006.

[85] Steven N. Minton, Mark D. Johnston, Andrew B. Philips, and Philip D. Laird. Minimizing conflicts: A heuristic repair method for constraint satisfaction and scheduling problems. *Artif. Intell.*, 58:161–205, 1992.

[86] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin A. Riedmiller, Andreas Fidjeland, Georg Ostrovski, Stig Petersen, Charlie Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518:529–533, 2015.

[87] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International Conference on Machine Learning*, 2016.

[88] Balas K. Natarajan. On planning assemblies. In *Proceedings of the Fourth Annual Symposium on Computational Geometry*, SCG '88, page 299–308, New York, NY, USA, 1988. Association for Computing Machinery. ISBN 0897912705. doi: 10.1145/73393.73424.

[89] Miguel Neves and Pedro Neto. Deep reinforcement learning applied to an assembly sequence planning problem with user preferences. *The International Journal of Advanced Manufacturing Technology*, 122(11):4235–4245, 2022.

[90] Nils J. Nilsson. *Principles of Artificial Intelligence*. Symbolic Computation. Springer Berlin, Heidelberg, 1 edition, 5 1982. ISBN 978-3-540-11340-9. Jointly published with Tioga Publishing Company.

[91] Diego Perez, Sanaz Mostaghim, Spyridon Samothrakis, and Simon M. Lucas. Multiobjective monte carlo tree search for real-time games. *IEEE Transactions on Computational Intelligence and AI in Games*, 7(4), 2015. doi: 10.1109/TCIAIG.2014.2345842.

[92] Pearl Pu and Lisa Purvis. Assembly planning using case adaptation methods. *Proceedings of 1995 IEEE International Conference on Robotics and Automation*, 1: 982–987 vol.1, 1995.

[93] Pearl Pu and Markus Reschberger. Assembly sequence planning using case-based reasoning techniques. *Knowledge-Based Systems*, 4(3):123–130, 1991. ISSN 0950-7051.

[94] Charles Ruizhongtai Qi, Li Yi, Hao Su, and Leonidas J Guibas. Pointnet++: Deep hierarchical feature learning on point sets in a metric space. volume 30, 2017.

[95] William C. Regli, Cheryl Foster, Erik Hayes, Cheuk Yiu Ip, David McWherter, Mitchell Peabody, Yuriy Shapirsteyn, and Vera Zaychik. National design repository project: a status report. In *International Joint Conferences on Artificial Intelligence (IJCAI), Seattle, WA*, pages 4–10. Citeseer, 2001.

[96] Ismael Rodríguez, Korbinian Nottensteiner, Daniel Leidner, Michael Kaßecker, Freek Stulp, and Alin Albu-Schäffer. Iteratively refined feasibility checks in robotic

assembly sequence planning. *IEEE Robotics and Automation Letters*, 4(2):1416–1423, 2019. doi: 10.1109/LRA.2019.2895845.

[97] Ismael Rodríguez, Korbinian Nottensteiner, Daniel Leidner, Maximilian Durner, Freek Stulp, and Alin Albu-Schäffer. Pattern recognition for knowledge transfer in robotic assembly sequence planning. *IEEE Robotics and Automation Letters*, 5(2): 3666–3673, 2020. doi: 10.1109/LRA.2020.2979622.

[98] Marco Santochi and Gino Dini. Computer aided planning of assembly operations: the selection of assembly sequences. *Robotics and Computer-Integrated Manufacturing*, 9(6):439–446, 1992.

[99] Thiusius Rajeeth Savarimuthu, Anders Glent Buch, Christian Schlette, Nils Wantia, Jürgen Roßmann, David Martínez Martínez, G. Alenyà, Carme Torras, Ale Ude, Bojan Nemec, Aljaz Kramberger, Florentin Wörgötter, Eren Erdal Aksoy, Jeremie Papon, Simon Haller, Justus H. Piater, and Norbert Krüger. Teaching a robot the semantics of assembly tasks. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 48:670–692, 2018.

[100] Ivan Serina. Kernel functions for case-based planning. *Artif. Intell.*, 174:1369–1406, 2010.

[101] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, L. Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Vedavyas Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy P. Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484–489, 2016.

[102] Balakumar Sundaralingam, Siva Kumar Sastry Hari, Adam Fishman, Caelan Garrett, Karl Van Wyk, Valts Blukis, Alexander Millane, Helen Oleynikova, Ankur Handa, Fabio Ramos, Nathan Ratliff, and Dieter Fox. curobo: Parallelized collision-free minimum-jerk robot motion generation, 2023.

[103] Sujay Sundaram, Ian Remmler, and Nancy M. Amato. Disassembly sequencing using a motion planning approach. *Proceedings 2001 ICRA. IEEE International Conference on Robotics and Automation (Cat. No.01CH37164)*, 2:1475–1480 vol.2, 2001.

[104] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT press, 2nd edition, 2018. ISBN 9780262039246.

[105] Arun Swaminathan and K. Suzanne Barber. An experience-based assembly sequence planner for mechanical assemblies. *IEEE Transactions on Robotics and Automation*, 12(2):252–267, 1996.

[106] Ulrike Thomas, Mark Barrenscheen, and Friedrich M. Wahl. Efficient assembly sequence planning using stereographical projections of c-space obstacles. *Proceedings of the IEEE International Symposium on Assembly and Task Planning, 2003.*, pages 96–102, 2003.

[107] Yunsheng Tian, Jie Xu, Yichen Li, Jieliang Luo, Shinjiro Sueda, Hui Li, Karl D. D. Willis, and Wojciech Matusik. Assemble them all. *ACM Transactions on Graphics (TOG)*, 41:1 – 11, 2022.

[108] Yunsheng Tian, Karl D. D. Willis, Bassel Al Omari, Jieliang Luo, Pingchuan Ma, Yichen Li, Farhad Javid, Edward Gu, Joshua Jacob, Shinjiro Sueda, Hui Li, Sachin Chitta, and Wojciech Matusik. Asap: Automated sequence planning for complex robotic assembly with physical feasibility. *2024 IEEE International Conference on Robotics and Automation (ICRA)*, pages 4380–4386, 2023.

[109] Carmelo Del Valle, Rafael M Gasca, Miguel Toro, and Eduardo F Camacho. A genetic algorithm for assembly sequence planning. In *Artificial Neural Nets Problem Solving Methods: 7th International Work-Conference on Artificial and Natural Neural Networks, IWANN2003 Maó, Menorca, Spain, June 3–6, 2003 Proceedings, Part II 7*, pages 337–344. Springer, 2003.

[110] Ashish Vaswani, Noam M. Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Neural Information Processing Systems*, 2017.

[111] Petar Velickovic, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lió, and Yoshua Bengio. Graph attention networks. *ArXiv*, abs/1710.10903, 2017.

[112] Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. Pointer networks. *ArXiv*, abs/1506.03134, 2015.

[113] Tom Vodopivec, Spyridon Samothrakis, and Branko Ster. On monte carlo tree search and reinforcement learning. *J. Artif. Intell. Res.*, 60, 2017.

[114] Weiwei Wan, Kensuke Harada, and Kazuyuki Nagata. Assembly sequence planning for motion planning. *ArXiv*, abs/1609.03108, 2016.

[115] Yue Wang, Yanmei Jiao, Rong Xiong, Hongsheng Yu, Jiafan Zhang, and Yong Liu. Masd: A multimodal assembly skill decoding system for robot programming by demonstration. *IEEE Transactions on Automation Science and Engineering*, 15 (4):1722–1734, 2018.

[116] Keijiro Watanabe and Shuhei Inada. Search algorithm of the assembly sequence of products by using past learning results. *International Journal of Production Economics*, 226:107615, 2020.

[117] Karl D. D. Willis, Pradeep Kumar Jayaraman, Hang Chu, Yunsheng Tian, Yifei Li, Daniele Grandi, Aditya Sanghi, Linh-Tam Tran, J. Lambourne, Armando Solar-Lezama, and Wojciech Matusik. Joinable: Learning bottom-up assembly of parametric cad joints. *2022 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 15828–15839, 2021.

[118] Randall H. Wilson and Jean-Claude Latombe. Geometric reasoning about mechanical assembly. *Artificial Intelligence*, 71(2):371–396, 1994.

[119] Randall H. Wilson, Lydia E. Kavraki, Tomas Lozano-Perez, and Jean-Claude Latombe. Two-handed assembly sequencing. *The International Journal of Robotics Research*, 14:335 – 350, 1995.

[120] Jan D. Wolter. On the automatic generation of assembly plans. *Proceedings, 1989 International Conference on Robotics and Automation*, pages 62–68 vol.1, 1989.

[121] Jan D. Wolter. A combinatorial analysis of enumerative data structures for assembly planning. In *Proceedings. 1991 IEEE International Conference on Robotics and Automation*, pages 611–618 vol.1, 1991. doi: 10.1109/ROBOT.1991.131649.

[122] Qinghua Wu, Xuejun Zhang, and Shan Guan. An assembly sequence planning method based on discrete difference genetic algorithm. In *2019 International Conference on Artificial Intelligence and Advanced Manufacturing (AIAM)*, pages 557–561. IEEE, 2019.

[123] Xiaoyang Wu, Li Jiang, Peng-Shuai Wang, Zhijian Liu, Xihui Liu, Yu Qiao, Wanli Ouyang, Tong He, and Hengshuang Zhao. Point transformer v3: Simpler, faster, stronger. *2024 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 4840–4851, 2023.

[124] Yasuyoshi Yokokohji, Yoshihiro Kawai, Mizuho Shibata, Yasumichi Aiyama, Shinya Kotosaka, Wataru Uemura, Akio Noda, Hiroki Dobashi, Takeshi Sakaguchi, and Kazuhito Yokoi. Assembly challenge: a robot competition of the industrial robotics category, world robot summit–summary of the pre-competition in 2018. *Advanced Robotics*, 33(17):876–899, 2019.

[125] Minghui Zhao, Xian Guo, Xuebo Zhang, Yongchun Fang, and Yongsheng Ou. Aspw-drl: assembly sequence planning for workpieces via a deep reinforcement learning approach. *Assembly Automation*, 40(1):65–75, 2020. doi: 10.1108/AA-11-2018-0211.

[126] Yin Zhou and Oncel Tuzel. Voxelnet: End-to-end learning for point cloud based 3d object detection. *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 4490–4499, 2017.

[127] Zhongxiang Zhou, Rong Xiong, Zexi Chen, and Yue Wang. Assembly sequence generation for new objects via experience learned from similar object. In *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 1054–1061, 2021. doi: 10.1109/IROS51168.2021.9636531.

[128] Eckart Zitzler. *Evolutionary algorithms for multiobjective optimization: Methods and applications*. Berichte aus der Informatik. Shaker Verlag, December 1999. ISBN 978-3-8265-6831-2.

Acronyms

AbD Assembly-by-Disassembly. v, vii, ix, 1, 2, 17, 24, 28, 29, 34, 35, 41–44, 48–50, 53, 54, 62, 87, 88, 90, 103, 109

ASP assembly sequence planning. v, ix, x, 1–5, 7, 9, 11, 12, 14, 15, 17, 19–32, 35–37, 39–43, 49, 53, 55, 60–62, 65, 67, 73–75, 79–81, 83, 87–91, 103, 107

BFS breadth-first search. 21, 22, 29, 44, 103

C-space configuration space. 19, 20, 103

CAD Computer-Aided Design. 7, 12, 15, 16, 27, 32, 38, 39, 103

CBR case-based reasoning. 7, 12, 14, 15, 27, 103

CNN convolutional neural network. 25, 103

CSP constraint satisfaction problem. 15, 103

DAG directed acyclic graph. 8–10, 15, 29, 103

DBG directional blocking graph. 18, 19, 103, 107

DFS depth-first search. 17, 21, 44, 62, 65, 103

DG disassembly graph. 44, 65, 103

DQL Deep Q-learning. v, vii, x, 4, 5, 23, 24, 32–34, 48, 62, 65, 67–69, 74, 85, 88, 91, 103

EST Expansive Space Tree. 21, 22, 103

EVT Expansive Voronoi Tree. 21, 22, 103

FNN Feed-Forward Neural Network. 32–34, 49, 68, 84, 103

GA Genetic Algorithm. 7, 29–32, 34, 36, 61, 103

GAT Graph Attention Network. 25, 27, 103

GNN Graph Neural Network. v, vii, x, 3–5, 13, 25, 36, 42, 46, 48–51, 53, 54, 62, 65, 67–69, 74, 79, 84, 85, 87, 88, 91, 103

GVD General Voronoi Diagram. 21, 22, 103

I-ML-RRT Iterative-ML-RRT. 21, 103

IAR-IPR Institute for Anthropomatics and Robotics - Intelligent Process Control and Robotics. 103

IK inverse kinematics. 64, 103

LCD Local Constraint Digraph. 17, 18, 103

LfD Learning from Demonstration. 7, 12, 14, 27, 103

M-space motion space. 19, 103

MCTS Monte Carlo Tree Search. v, vii, viii, x, 3–5, 35–37, 62, 63, 65–72, 74, 79–86, 88, 89, 91, 103, 107, 109

MDP Markov Decision Process. x, 22–24, 48, 62, 65, 67, 74, 79, 103

ML-RRT Manhattan-like RRT. 20, 21, 103

MOUCB Multi-Objective Upper Confidence Bound. 82, 83, 103

MSE mean squared error. 103

NDBG Non-directional blocking graph. 18, 19, 27, 103, 107

NSGA-II Non-dominated Sorting Genetic Algorithm II. 32, 37, 103

PRM probabilistic roadmap. 19, 20, 27, 103

RL Reinforcement Learning. 4, 7, 23, 25, 29, 30, 32, 33, 36, 48, 61, 103

RRT Rapidly-exploring Random Tree. 19–22, 27, 44, 103

UCT Upper Confidence Bounds for Trees. 66, 82, 103

List of Figures

List of Tables