2024 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)
October 14-18, 2024. Abu Dhabi, UAE

Beyond Feasibility: Efficiently Planning Robotic Assembly Sequences That Minimize Assembly Path Lengths

Alexander Cebulla, Tamim Asfour, and Torsten Kröger

***Abstract*— Advancements in Industry 4.0 demand sophisticated solutions for automatic robotic assembly sequence planning (RASP), capable of handling the diversity and complexity of modern manufacturing tasks. One approach to RASP is Assembly-by-Disassembly (AbD). It first searches for a disassembly sequence that is then inverted to obtain an assembly sequence. One of the challenges of AbD, however, is the exponential number of potential assembly sequences for any given assembly. To mitigate this challenge, we propose to transfer knowledge obtained during previous planning attempts. Specifically, we present an approach that combines Monte Carlo Tree Search (MCTS) with deep Q-learning to optimize the total length of robotic assembly paths. We use a graph-based representation of disassembly states in combination with a graph neural network to learn the Q-function. We further discuss a principled approach to generate 3D assemblies out of aluminium profiles that a single robot manipulator can assemble. With this approach, we generated two datasets consisting of 14 assemblies with 21 removable parts and 7 assemblies with 30 removable parts. Using leave-one-out cross-validation, we were able to demonstrate how our approach outperformed an unmodified MCTS. Moreover, we successfully transferred knowledge between datasets.**

I. Introduction

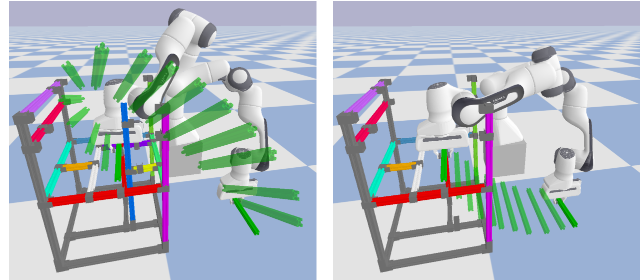
With the ongoing transition to Industry 4.0, an increasing number of manufacturers seek to develop customized product variants [1]. An important step towards this goal is the automation of robotic assembly sequence planning (RASP), which specifies the sequence for assembling individual parts into a finished product with a robotic manipulator.

While finding a feasible assembly sequence for a given product automatically is already a challenging task, it also should not be an arbitrary assembly sequence, but one that allows the robot to assemble the product fast and efficiently. These rather broad terms can be measured by several factors: The actual time that is needed to complete an assembly, the energy consumption during this process, and the length of the path taken during assembly, which can also influence both aforementioned factors.

Our goal is to find a feasible assembly sequence for a given product that is optimized with respect to the length of the employed assembly paths, which tend to facilitate short assembly times and energy efficiency. However, for a given product with N part, $N!$ potential assembly sequences exist - a combinatorial explosion. Therefore, a method that investigates only a subset of potential solutions in a guided way is necessary, leading to near-optimal solutions within a fraction of computation time.

The research leading to these results has received funding from the Carl Zeiss Foundation through the AgiProbot project and the German Research Foundation (DFG) within the SFB 1574.

Institute for Anthropomatics and Robotics - Intelligent Process Automation and Robotics Lab (IAR-IPR), Karlsruhe Institute of Technology (KIT) {alexander.cebulla, asfour, torsten}@kit.edu.



(a) The part must be moved along a longer path over the assembly.

(b) The part can be moved along a shorter path through the inside of the assembly.

Fig. 1: Two assembly paths were planned for the same part, but at different steps in the assembly sequence.

One established approach for discovering feasible assembly sequences is assembly-by-disassembly (AbD), which starts with the assembled product and tries to disassemble it step by step. Under the assumption that assembly and disassembly are invertible, it then inverts this sequence. We propose to guide AbD towards feasible and efficient (dis)assembly sequences. Specifically, we employ a Monte Carlo tree search (MCTS), together with a learned Q-function as a heuristic for finding promising parts for removal. The Q-function is implemented through a Graph Neural Network (GNN), that works based on the current state of the assembly, where parts are presented as nodes and existing connections as edges. It is trained with the goal of finding removable parts such that the sum of assembly path lengths for the full assembly sequence is minimized.

We also discuss a principled approach for creating 3D assemblies out of aluminium profiles that can be assembled by a single robot manipulator as shown by Fig. 1. We created two sets of assemblies: the first one consists of 14 assemblies with 21 removable parts and the second one of 7 assemblies with 30 removable parts. We evaluated our approach on these two sets using leave-one-out cross-validation and, when compared against an unmodified MCTS, could produce assembly sequences with total path lengths that were on average 0.8 m shorter for the first and 1.4 m shorter for the second dataset. When training on the first dataset and testing on the second, we observed that the total path lengths of sequences were, on average, 1.6 m shorter. This result demonstrates the successful transfer of knowledge across varying levels of assembly complexity.

II. Related Work

Given the significant cost-saving opportunities and the intricate challenges it presents, the optimization of assembly sequences has received extensive research [2]. In [3], AND/OR graphs were introduced to represent all feasible assembly sequences. Based on this, a graph search was used to optimize assembly sequences with respect to three objectives. This approach was extended in [4], which described an anytime search algorithm for optimizing against various objectives. In [5] an efficient method for creating disassembly graphs was discussed. The main idea was to first use a fast, but less accurate test for feasible part removal. If that failed, a slower but more accurate test was used. In the disassembly graph, each node represented a disassembly state and each edge the removal of a specific part. A depth-first search and a breadth-first search were then applied to find feasible assembly sequences. To find optimal assembly sequences, given such a disassembly graph, an alternative approach based on Ant Colony Optimization (ACO) was suggested in [6]. Further, in [7] the max–min ant colony system was introduced as an extension to ACO and demonstrated to be more efficient when compared against various other optimization algorithms commonly used for ASP such as particle swarm optimization (PSO) or genetic algorithm (GA). A discrete version of PSO was used in [8] to minimize assembly costs while guaranteeing feasible assembly sequences. An ASP framework, based on a multi-objective GA, was introduced in [9] to find Pareto-optimal sequences for assemblies with deformable parts. This is extended in [10], where it is demonstrated how this framework can be used in a human-robot-collaboration context, i.e., for finding robotic disassembly sequences where some steps are executed by humans.

The works discussed so far all have in common that they start from scratch for each new assembly. This leads to a high time complexity because there are $N!$ potential assembly sequences for an assembly with N parts. Additionally, checking the feasibility of each step in these sequences requires extensive computation. To mitigate this computational overhead, recent works have leveraged graph-based representations of the assembly in combination with deep learning to either directly predict feasible sequences [11], [12] or only compute feasibility checks when there is a high likelihood of them being successful [13]. However, these works solely focused on finding feasible sequences.

In contrast, in this work, using the AbD approach to ASP, we want to identify parts that are not only feasible to remove, but will also minimize the expected path length for the overall sequence. We propose to use a Q-function trained on previously planned assembly sequences to guide an MCTS to promising areas in the search space. This approach has successfully been applied [14] to handle large search spaces such as those encountered in games like Go and chess. In [15] it has successfully been applied to assemble wooden blocks of varying shapes such that they fill out a predefined volume with minimal deviation. Furthermore, in previous work [16], we demonstrated the feasibility of this approach for efficiently finding assembly sequences that reduced the number of direction changes required to assemble a part, while increasing its accessibility. However, we did not consider constraints imposed by a robotic manipulator. In fact, the computation of features for parts and their interrelations relied on so-called distance maps [17], which require that each part can be assembled along a straight line, while their orientations remain constant. This is an unreasonable assumption for many assemblies where parts need to be rotated and moved around other parts. In this work, we remove this assumption and allow arbitrary removal paths.

III. Problem Statement

We address the problem of efficiently finding an assembly sequence, such that the total length of all the paths that a robotic manipulator must execute to assemble all parts is minimized. Using the AbD assumption that assembly and disassembly are the inverse of each other, we will search for a disassembly sequence that optimizes this criterion. To formalize the problem, we define an assembly consisting of N parts as a set $\mathcal{A} = \{p_1, p_2, \ldots, p_N\}$, with each part p_i being characterized by its pose and geometric shape. Also, each part p_i has a set of grasp points $\mathcal{G}_i$ assigned to it, where each grasp point $g_{i,j} = (\boldsymbol{\tau}, \boldsymbol{\Gamma}, \omega)$ is defined by its center point $\boldsymbol{\tau} \in \mathbb{R}^3$, the orientation of the gripper $\boldsymbol{\Gamma} \in \mathrm{SO}(3)$, and the width of the gripper's fingers ω, where we dropped the indices for readability. Additionally, we define a removal path $q_{g_{i,j}} = (\boldsymbol{H}_0, \boldsymbol{H}_1, \ldots, \boldsymbol{H}_M)$ for a robotic gripper that has grasped part p_i at grasp point $g_{i,j}$ as a sequence of gripper poses $\boldsymbol{H} \in \mathrm{SE}(3)$ relative to the world coordinate system. A robotic disassembly sequence is then defined by a sequence of removal paths $\boldsymbol{u} \in \mathcal{U}$, where $\mathcal{U}$ is the set of all feasible sequences. That is, each path $q_{g_{i,j}} \in \boldsymbol{u}$ allows a robotic gripper to remove part p_i after grasping it at grasp point $g_{i,j}$ without causing collisions and while maintaining the assembly's stability throughout the removal process.

Let now

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

be the Euclidean distance between the translational components of the gripper poses. The length of a removal path $q_{g_{i,j}}$ consisting of M gripper poses is then given by

$$l(q_{g_{i,j}}) = \sum_{k=0}^{M-1} d(\boldsymbol{H}_k, \boldsymbol{H}_{k+1}). \tag{2}$$

We now define a loss function for a robotic disassembly sequence $\boldsymbol{u}$ as the sum of its removal paths' lengths

$$\mathcal{L}(\boldsymbol{u}) = \sum_{q_{g_{i,j}} \in \boldsymbol{u}} l(q_{g_{i,j}}). \tag{3}$$

The overall problem is then given by

$$\min_{\boldsymbol{u} \in \mathcal{U}} \mathcal{L}(\boldsymbol{u})$$

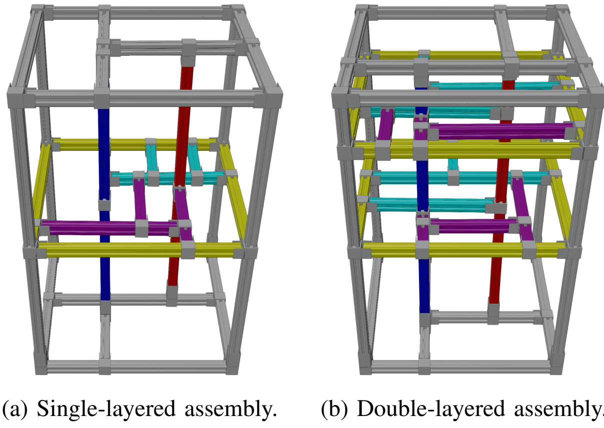

(a) Single-layered assembly. (b) Double-layered assembly.

Fig. 2: Each layer consists of a surrounding frame (yellow), and profiles in the back (cyan) and the front (purple).

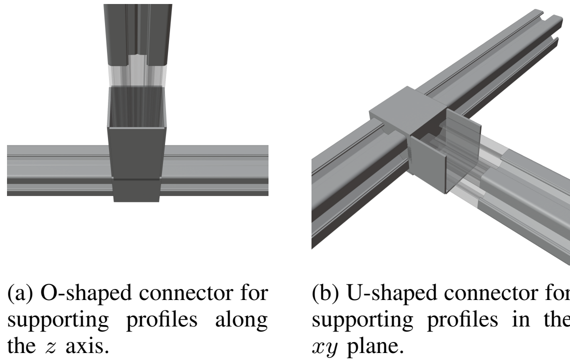
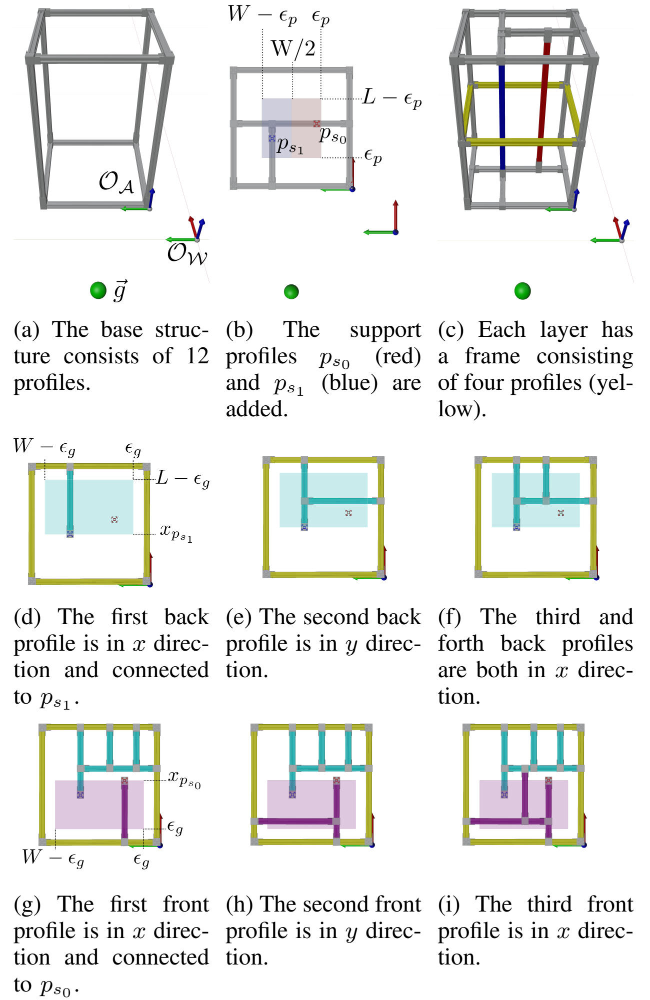

(a) O-shaped connector for supporting profiles along the z axis. (b) U-shaped connector for supporting profiles in the xy plane.

Fig. 3: U- and O-shaped connectors.



(a) The base structure consists of 12 profiles. (b) The support profiles p_{s_0} (red) and p_{s_1} (blue) are added. (c) Each layer has a frame consisting of four profiles (yellow).

(d) The first back profile is in x direction and connected to p_{s_1}. (e) The second back profile is in y direction. (f) The third and forth back profiles are both in x direction.

(g) The first front profile is in x direction and connected to p_{s_0}. (h) The second front profile is in y direction. (i) The third front profile is in x direction.

Fig. 4: Creation process of the assembly shown in Fig. 2a. In the upper row the goal position $\vec{g}$ as well as the world coordinate system and the assembly coordinate system $\mathcal{O}_\mathcal{W}$ and $\mathcal{O}_\mathcal{A}$, respectively, are shown. For the remaining steps, we only show $\mathcal{O}_\mathcal{A}$.

IV. Aluminium Profile Assemblies

Several recent works [12], [13], [18], [19] have used assemblies made out of aluminium profiles for evaluating ASP approaches. In particular, [12] and [18] generated 2D assemblies consisting of 3 to 10 parts, designed to be assembled on a table top. Inspired by these works, we implemented a generator to create 3D assemblies out of 20mm×20mm aluminium profiles, as shown in Fig. 2.

Our goal was that each assembly could be assembled by a single robotic manipulator. We, thus, used two types of connectors. The first type is the U-shaped connector shown in Fig. 3b, which is used for profiles in the xy plane of the world coordinate system $\mathcal{O}_\mathcal{W}$. The second type is the O-shaped connector shown in Fig. 3a, which we used for profiles oriented along the z axis of $\mathcal{O}_\mathcal{W}$. The world coordinate system $\mathcal{O}_\mathcal{W}$ is depicted in Fig. 4a. A connector is added if the short side of a profile comes in contact with the long side of a profile.

In the following, we provide a detailed description of how such assemblies were generated and how we planned removal paths for individual profiles.

A. Generating an Assembly

All generated assemblies share a general layout of base structure; support profiles and layers of profiles attached to the base structure and the support profiles. We will now describe each of these parts in detail. An overview of the construction of the assembly depicted in Fig 2a is shown in Fig. 4.

Each assembly consists of several aluminium profiles. As shown in Fig. 2 and Fig. 4 each one is oriented along one of the axes of $\mathcal{O}_\mathcal{W}$. For clarity, we will omit $\mathcal{O}_\mathcal{W}$, when describing the orientation of a profile.

The construction starts with a base structure, shown in Fig. 4a, that consists of 12 profiles. Specifically, for the bottom and top layer of the assembly four profiles each, arranged as a square, are used. These two squares are then connected by four profiles oriented along the z axis. All other profiles will be inside the cuboid formed by the base structure, thus, its height H, width W and length L define the dimensions of the overall assembly.

Next, we add two profiles, p_{s_0} and p_{s_1}, oriented along the z axis, as illustrated in the top view presented in Fig. 4b. We will also refer to them as support profiles. Their x position in the assembly's coordinate system $\mathcal{O}_\mathcal{A}$ is randomly selected

between ϵ_p and the length of the assembly L minus ϵ_p, where ϵ_p is the minimum length a profile is required to have. That is, $\epsilon_p < x_{p_{s_0}}, x_{p_{s_1}} < L - \epsilon_p$. Further, their y positions are randomly chosen such that $\epsilon_p < y_{p_{s_0}} < W/2$ and $W/2 < y_{p_{s_1}} < W - \epsilon_p$, where W the width of the assembly. As shown in Fig. 4c, p_{s_0} and p_{s_1} are connected to the assembly's bottom and top through profiles, which lie in the xy plane and have their orientation randomly determined.

We, then, add layers consisting of profiles either oriented along the x or y axis, which are connected to the base structure and the support beams. Each layer is created following three steps:

a) Frame creation: We add a frame, shown in yellow in Fig. 4c, consisting of four profiles in the xy plane that are connected to the base structure.

b) Creating the back: We add profiles to the back of the layer as depicted in Fig. 4d to Fig. 4f. A profile is considered at the back of the layer if it is either partially or completely behind the support profile that is closer to the front. In detail, "behind" and "front" are defined from the perspective of the goal position $\vec{g}$, which is visualized by a green sphere in Fig. 4. For instance, in Fig.4b and Fig. 4c, the blue support profile p_{s_1} is positioned closer to the front. The goal position $\vec{g}$ corresponds to the translational component of the final pose $\boldsymbol{H}_M$ of all removal paths, as defined in Sec. III. Intuitively, if a to-be removed profile p_i is either obscured by or close to the "shadow" of another profile p_j – as observed from $\vec{g}$ – a robotic manipulator must most likely navigate around p_j while moving p_i to $\vec{g}$.

We add profiles iteratively, starting by randomly selecting either the x or y orientation for each new profile. If a support profile has no connected profiles yet, we randomly determine whether the new profile should be connected to a support profile. In cases where a connection to a support profile is decided, the next steps are to choose which support profile to connect to and then, based on the initially selected direction, set either the x or y coordinate of the new profile accordingly. For example, in Fig. 4d the first profile is in x direction and connected to p_{s_1}.

If connecting to a support profile is not selected, the positioning of the new profile depends on its direction: for profiles oriented in the y direction, we randomly select an x coordinate such that $x_{p_{s_i}} < x < L - \epsilon_g$, where $x_{p_{s_i}}$ is the x coordinate of the support profile closer to the front, in our case $i = 1$. Conversely, for profiles in the x direction, we choose a y coordinate with $\epsilon_g < y < W - \epsilon_g$. Hereby, ϵ_g is the minimum distance that we require between parallel profiles such that they can be grasped by the robotic manipulator. This was done to have a higher probability of a layer profile being connected to a support profile, while still allowing for layers without such connections. After the direction and the corresponding coordinate have been determined, we first check if the distance between the new profile and any profiles that have the same direction is smaller than ϵ_g. In this case, the profile is not added and the above steps are repeated. Otherwise, we try to add the profile with its length set to either the assembly's width W or length L depending on its direction. We test if it is intersected by any other already added profiles. If this is not the case, we directly add it. Otherwise, we split it into segments. We, then, remove all segments that either lie partially or completely ahead of the support profile in the front or are shorter than ϵ_p, that is, a profile's minimum length. Finally, we randomly select one of the segments and add it.

c) Creating the front: During the third step, we add profiles to the front of the layer as shown in Fig. 4g to Fig. 4i. The process is similar to the previous step with the difference that we use the support profile $x_{p_{s_j}}$ in the back as the new reference point. That is, the x coordinates are now selected such that $\epsilon_g < x < x_{p_{s_j}}$. In our example, $j = 0$.

B. Disassembling a Profile With a Robotic Manipulator

Given a profile $p_i \in \mathcal{A}$, we test if it can be removed from the assembly using a hierarchical approach: First, we test if it is constrained by any other profile. For that, we move it up, i.e., along the z direction, by 5 cm and check for collisions. In case there are none, we next test for all grasp points $g_{i,j} = (\boldsymbol{\tau}, \boldsymbol{\Gamma}, \omega) \in \mathcal{G}_i$ if they are reachable. That is, we try to compute the inverse kinematic (IK) for the robotic gripper such that its orientation is $\boldsymbol{\Gamma}$, the midpoint between its fingers corresponds to the grasp point's center $\boldsymbol{\tau}$ and the width of the gripper's fingers is ω. If the algorithm cannot find a solution or the solution is in collision we ignore the grasp point as unreachable. Let now $\mathcal{G}_i^r$ be the set of all reachable grasp points. Then, for each reachable grasp point $g_{i,j}^r \in \mathcal{G}_i^r$, we compute a removal path $q_{g_{i,j}^r}$. If a collision-free removal path could be found, we identify p_i as removable. We denote the shortest collision-free removal path as $q^s_{g_{i,j}^r}$. Formally, $\forall g_{i,j}^r, g_{i,k}^r \in \mathcal{G}_i^r; j \neq k; l(q^s_{g_{i,j}^r}) \leq l(q_{g_{i,k}^r})$, where $l(.)$ is the length of a removal path as defined in Eq. 2. To enhance computational efficiency, the computation of removal paths for each reachable grasp point is conducted in parallel using the cuRobo [20] robot motion planning library. Its pipeline integrates an IK solver, a sampling-based planner, and an L-BFGS optimizer, delivering paths optimized for both length and smoothness.

V. Efficient Robotic Assembly Sequence Planning via Guided Monte Carlo Tree Search

We use an MCTS in combination with an AbD strategy to find robotic assembly sequences that are optimized with respect to the assembly path length. That is, the search starts from a fully assembled product and iteratively tries to remove parts in simulation with a robotic manipulator until all parts are removed. If a feasible disassembly sequence is found it is inverted to obtain the assembly sequences. MCTS uses a tree policy – often based on the Upper Confidence bounds for Trees (UCT) [21] – which relies on data from previous searches to guide the exploration. However, due to the combinatorial explosion – there are $N!$ potential sequences for an assembly with N parts – finding optimal solutions is time-consuming. As an alternative approach, we propose to train a Q-function on previous search results to guide the exploration of an MCTS for a novel assembly.

In the following, we first formalize RASP as a Markov decision process (MDP). Next, we describe the MCTS and how we applied it to RASP. Then, we introduce a graph representation for (dis)assembly states and how we used this representation to train a GNN to approximate a Q-function. Finally, we discuss how we guided the exploration of the MCTS with this function.

A. Modelling Robotic Assembly Sequence Planning as Markov Decision Process

A MDP is a tuple (S, A, P, R), where S represents the set of states and A denotes the set of actions. The probability of transitioning to state s_{t+1} after executing action a_t in state s_t is then $P(s_t, a_t, s_{t+1})$. During this transition an immediate reward $R(s_t, a_t, s_{t+1})$ is obtained.

We model the RASP via AbD process as a finite MDP, where, at each step, the primary decision is to determine the next part to be removed. We denote S as the set of all disassembly states $s \subseteq \mathcal{A}$, with each disassembly state comprising the parts that are still in the assembly. An action a_t corresponds to the removal of a part p_i along the shortest removal path, i.e., $a_t = q^s_{g^r_{i,j}}$ as defined in Sec. IV-B. Considering that the disassembly process in this context is deterministic, the transition probability $P(s_t, a_t, s_{t+1})$ equals 1 for all feasible transitions. That is, removal path $a_t = q^s_{g^r_{i,j}}$ is collision-free, and the resultant disassembly state s_{t+1} remains stable under gravity. Finally, we define the immediate reward as the negative value of $l(q_{g_{i,j}})$, which is the removal path's length as defined in Eq. 2. That is, the reward is

$$R(s_t, a_t, s_{t+1}) = -l(q_{g_{i,j}}).$$

Decision-making is modeled via a policy function $\pi : S \rightarrow A$ that maps states to actions. Given a policy, we compute its expected reward as

$$\mathbb{E}\left[\sum_{t=0}^{N-1} \gamma^t R(s_t, a_t, s_{t+1})\right]; a_t = \pi(s_t), \tag{4}$$

where $0 \leq \gamma \leq 1$ is a discount factor and N the number of parts in assembly $\mathcal{A}$. The goal is then to find an optimal policy π^* that has the maximum expected reward.

B. Applying Monte Carlo Tree Search to Robotic Assembly Sequence Planning

Building upon the MDP framework outlined in Sec.V-A, our approach employs an MCTS [22] to identify optimal assembly sequences. That is, they maximize the expected reward defined in Eq. 4 and thus minimize the loss function defined by Eq. 3.

The MCTS constructs a search tree, where each node, denoted as n^T, encapsulates a specific disassembly state $s \subseteq \mathcal{A}$. Further, each node also tracks how often it was visited $N^T(n^T)$. Commonly [22], the sum of the cumulative rewards obtained from the subsequent disassembly sequences starting from the current state s is also recorded and then used to compute the average expected reward. However, we instead decided to track the maximum overall cumulative rewards Q^T. As discussed in Sec. IV-B, we used a motion planning pipeline [20] that optimizes paths for length. Still, we observed some variability in planned path lengths. By tracking the maximum instead of the average expected reward, we reduced the influence of outliers.

To build the tree, it iteratively executes the following four steps until a search budget is exhausted:

1) **Selection**: This step involves navigating the tree from the root to an unexpanded leaf node using a tree policy. To realize this policy, we used a slightly modified version of the Upper Confidence bounds for Trees (UCT) [21] taking into account that we tracked the maximum instead of the average expected reward:
$$UCT(n^T, n'^T) = \hat{Q}^T(n'^T) + c\sqrt{\frac{2 * \ln N^T(n^T)}{N^T(n'^T)}}$$
$$n'^T = \underset{n'^T \in \text{Children}(n^T)}{\text{argmax}} UCB(n^T, n'^T),$$
where n'^T is a child of n^T, and $\hat{Q}^T$ represents the normalized maximum reward observed, scaled between 0 and 1. This normalization, suggested in [23], is done using the maximum and minimum Q^T values of a node's children as bounds. For new nodes without local data, we used the global maximum and minimum values. Finally, c is a constant determining the balance between the exploration of less-visited nodes and the exploitation of nodes with high rewards.
2) **Expansion**: Once a leaf node is reached with unexplored children, one is randomly selected and added to the tree. In detail, the shortest collision-free removal path $q^s_{g^r_{i,j}}$ corresponding to the action required to go from the parent to the leaf node is planned, as discussed in IV-B, and stored in the parent node. To handle the deviations in path length exhibited by the motion planning library [20], we repeated this planning k times and then selected the overall shortest path.
3) **Simulation**: In this step, a simulation is performed from the newly added node to estimate the expected reward that can be obtained starting from its associated disassembly state. In detail, a default policy, here the random selection of viable actions, is used to simulate the disassembly process until all parts are removed. In contrast to the removal paths planned during node expansion, the removal paths planned during the simulation step are not permanently stored. Therefore, to reduce computation time, we only searched once for the shortest removal path during simulation steps and relied on our tracking of the maximum reward, as discussed above, to mitigate the variability observed during simulation.
4) **Backpropagation**: After the simulation, the results are propagated back up the tree. Each node visited during the selection and expansion phases is updated with the new information, adjusting their estimated reward values Q^T and visit counts N^T.

After the allocated search budget is exhausted, a child node is selected using UCT, marking the completion of one search step within the current MCTS episode. If this node is not a leaf, we continue the episode and start the next search step. Otherwise, we end the current episode and start the next one from the root node.

C. Representing a Disassembly State as Graph

We represent a disassembly state $s \subseteq \mathcal{A}$ as a directed, fully-connect graph $G_{\mathcal{A}} = (N_{\mathcal{A}}, E_{\mathcal{A}})$, consisting of a set of nodes $N_{\mathcal{A}}$ and a set of edges $E_{\mathcal{A}}$. Each part $p_i \in s$ corresponds to a node $n_i \in N_{\mathcal{A}}$. Furthermore, to have a spatial representation of part p_i's dimensions and orientation, each node n_i has a node attribute $\hat{\boldsymbol{n}}_i$ that stores the eight corner points of the corresponding part p_i. The connections between parts p_i and part p_j are represented as directed edges $e_{i,j}, e_{j,i} \in E_{\mathcal{A}}$. To capture the spatial layout of the disassembly state s, each edge has an edge attribute $\hat{\boldsymbol{e}}_{i,j}$ associated with it, which stores a vector pointing from the midpoint of part p_i to the midpoint of part p_j.

D. Deep Q-learning with Graph Neural Networks

Deep Q-learning (DQL) [24] is another approach besides MCTS for finding an optimal policy π^* that will maximize the expected reward defined in Eq. 4. It first learns a Q-function – also called a state-action function – by training a neural network. This function estimates the expected reward for following a particular policy π after taking a specific action a in a state s. Given such a function, the optimal policy π^* can then be computed by selecting the action that maximizes the Q-value for the current state-action pair $\pi(s) = \arg\max_{a \in A} Q(s, a)$.

DQL stores experiences $(s_t, a_t, s_{t+1}, R(s_t, a_t, s_{t+1}))$ encountered throughout the exploration of the MDP in a replay buffer. During the learning phase, it samples random batches of these experiences to refine the neural network's parameters. This decouples sequential dependencies among experiences, which facilitates more robust learning. DQL operates as an offline reinforcement learning algorithm, meaning it leverages experiences that might have been collected under various policies, not necessarily the one currently being optimized. Over time, the learned policy improves, becoming better at identifying and executing actions that will maximize the expected rewards.

Based on our graph representation described in Sec. V-C, we used a GNN [25] to learn a Q-function, which is a specialized neural network architecture that can handle graphs of varying sizes. In detail, for a given layer l, a GNN Φ^l processes an input graph $G^l = (N^l, E^l)$ by executing three transformations: First, the attributes of each edge as well as the attributes of its connected nodes are processed by a dedicated feed-forward neural network (FNN) ϕ_e^l resulting in updated edge attributes. Then, to update the node attributes, attributes from their neighboring nodes and connecting edges are aggregated. Next, a second dedicated FNN ϕ_n^l processes these aggregated attributes combined with the to-be updated node attributes:

$$\begin{aligned}
\hat{\boldsymbol{e}}_{i,j}^{(l+1)} &= \phi_e^l(\hat{\boldsymbol{n}}_i^l, \hat{\boldsymbol{e}}_{i,j}^l, \hat{\boldsymbol{n}}_j^l) \\
\hat{\boldsymbol{n}}_i^{*,(l+1)} &= \sigma^l \left(\sum_{e_{i,j} \in E_{\mathcal{A}}} \mathrm{ReLU}\left(\hat{\boldsymbol{n}}_i^l \oplus \hat{\boldsymbol{e}}_{i,j}^{(l+1)} \right) \right) \\
\hat{\boldsymbol{n}}_i^{(l+1)} &= \phi_n^l \left(\hat{\boldsymbol{n}}_i^l, \hat{\boldsymbol{n}}_i^{*,(l+1)} \right),
\end{aligned}$$

where $\oplus$ denotes the concatenation operation, and $\sum$ denotes aggregation over all edges connected to node n_i. Also, as recommended by [26], we used a *SoftMax* σ^l for aggregation instead of other functions like *mean* or *max*.

The output of each layer is a modified graph that retains the structure of the input but has updated node and edge attributes. That is, for a GNN with L layers:

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

Following [16], we can now define the Q-function $Q(s_t, a_t)$ as follows:

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

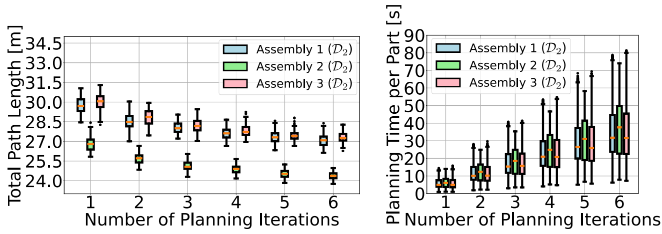
where ϕ_Q is a FNN that calculates the Q-value based on the state-action pair. Specifically, the state s_t is represented by the mean over all node attributes of nodes in $N_{s_t}^L$, that is the node set of the graph $G_{s_t}^L$ computed by the GNN's final layer. The action a_t is represented by the node attribute of node $n_{a_t}^L \in N_{s_t}^L$ which is the node that corresponds to the part that would be removed by a_t.

E. Guiding the Monte Carlo Tree Search with a learned Q-function

As stated in Sec. V-D, deep Q-learning is an offline algorithm. That is, we can use experiences collected during simulation steps of previous MCTS episodes on other assemblies to train a Q-function. We can then use this Q-function instead of the default policy described in Sec. V-B to guide the simulation. Additionally, we incorporate an ϵ-greedy strategy within our simulation steps. That is, during each step, a random value r is selected within the range $[0, 1]$. If $r > \epsilon$, the simulation uses the pre-trained Q-function, selecting the part associated with the highest predicted reward for removal. Otherwise, it reverts to the default policy, thus randomizing part selection. This ensures a balance between leveraging the predictive power of the pre-trained Q-function and maintaining a degree of randomness essential for discovering new assembly sequences.

VI. Creating the Datasets

To evaluate the effectiveness of our guided MCTS compared to a vanilla MCTS, we generated 14 single-layer assemblies with 21 removable profiles and 7 double-layer assemblies with 30 removable profiles as discussed in Sec. IV-A. That is, to save computation time, we did not disassemble the bottom profiles and the two profiles, oriented along the z axis, in the back of the base structure. These profiles are



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

(b) Time required per part for repeated planning.

Fig. 5: Results for the path length deviation analysis done on the first three double-layered assemblies.

shown in grey in Fig. 1. All assemblies have dimensions of $L = W = 40$ cm and $H = 60$ cm. Both – single- and double-layer assemblies – have two support profiles and three front profiles in either one or both layers. They differ in the number of profiles used in the back. While the double-layer assemblies have three profiles in the back of each layer, the single-layer assemblies have four profiles. Grasp points for each profile were systematically sampled at 2 cm intervals along all four sides, starting and ending 3 cm from the profile's end. An illustration of one assembly from each category is provided in Fig. 2. To train the Q-function, we collected data from both assembly types using the vanilla MCTS as described in Sec. V-B and Sec. V-E. Specifically, we recorded experiences from all MCTS simulations. This data was stored in two datasets, $\mathcal{D}_1$ for single-layer assemblies and $\mathcal{D}_2$ for double-layer assemblies.

VII. Experiments

We conducted three sets of experiments to evaluate our approach. Specifically, we had the following objectives:

1) Investigating the influence of the path planning algorithm [20] on the overall path length.
2) Understanding the benefits of using a pre-trained Q-function to guide MCTS for RASP when comparing it against vanilla MCTS.
3) Testing the generalizability of the trained Q-function across varying levels of assembly complexity.

For all MCTS experiments, we used a search budget of five. For the ϵ-greedy policy, we set $\epsilon = 0.2$

A. Path Length Deviation Analysis

Our first set of experiments aims at analyzing the impact the sampling-based path planner used by cuRobo [20] had on the removal path lengths. Due to its probabilistic nature, planning the removal of the same part in the same disassembly state can lead to multiple disassembly paths that deviate in length. While the cuRobo planning pipeline also optimizes the path length, this is done only locally. That is, if the sampling-based planner initially finds a longer path through the assembly, subsequent optimizations may only refine this path without exploring potentially shorter alternatives. In the following, we analyze how strong the effect on the overall path length is, and how well it can be mitigated by repeating

TABLE I: Comparison of the total removal path lengths found via vanilla MCTS and MCTS guided by a pre-trained Q-function for single-layer assemblies. The number of MCTS episodes was abbreviated as '#epi' and the number of search steps was abbreviated as '#st'. The shorter path length for each assembly is shown in bold.

	Vanilla MCTS for one-layered assemblies			MCTS + Q-function trained on $\mathcal{D}_1$	
id	**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)

the removal path planning multiple times and then choosing the shortest one.

We first searched for a feasible robotic disassembly sequence for each of the first three two-layered assemblies. Then, we repeatedly planned each removal path in the corresponding sequences between one and six times and selected the shortest path. This was repeated 50 times. The resulting overall path lengths are shown in Fig. 5a and the corresponding planning time per part in Fig. 5b. As shown in Fig. 5a for all three assemblies replanning reduces the total path length by around 3 m. At the same time, it increases the planning time per part by around 5 s for each additional iteration. Due to the minor improvements regarding path length compared to the increase in computation time when using more iterations, we decided to use four planning iterations, i.e., $k = 4$ as defined in Sec. V-B, during the MCTS expansion step for all remaining experiments.

B. Evaluating Guided MCTS on the Same Assembly Type

Initially, we focused on training and testing our model separately on single-layer (dataset $\mathcal{D}_1$) and double-layer (dataset $\mathcal{D}_2$) assemblies. In both cases, we used a leave-one-out cross-validation technique such that we tested on one assembly and trained on all the other assemblies of the same type.

a) Single-layer assemblies: As shown in Tbl. I, the guided MCTS, for all assemblies, achieved shorter path lengths compared to vanilla MCTS. A length reduction greater than 0.5 m, has been achieved for assemblies 4, 6, 7, 8, 9, 11, 13 and 14. For most assemblies, the best path length was found faster by the guided MCTS. In fact, for assemblies 7, 10 and 13 the best path length was found after

TABLE II: Comparison of the total removal path lengths found via vanilla MCTS and MCTS guided by a Q-function for double-layer assemblies. The number of MCTS episodes was abbreviated as '#epi' and the number of search steps was abbreviated as '#st'. The shortest path length for each assembly is shown in bold.

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)

the first search step, indicating that for these assemblies the use of the Q-function is sufficient to find a good solution.

b) Double-layer assemblies: The results are shown in Tbl. II. Similar to the single-layer experiments, the integration of a pre-trained Q-function consistently resulted in shorter path lengths when compared to the vanilla MCTS. A length reduction greater than 0.5 m was achieved for all assemblies except the first and the last one, where a path length similar to the one found by the vanilla MCTS was achieved.

C. Investigating the Generalizability of the Q-Function

We evaluated how a Q-function, trained exclusively on data from single-layer assemblies – i.e., on data from $\mathcal{D}_1$ – would perform when applied to double-layered assemblies. The goal was to test the generalizability of the trained Q-function across varying levels of assembly complexity. These results are shown in Tbl. II. For all assemblies, except 6, shorter path lengths were obtained when compared to the vanilla MCTS. For assemblies 1, 4 and 5 a reduction in path length greater than 0.5 m was achieved when compared to those paths found by the Q-function trained on $\mathcal{D}_2$. Finally, for all assemblies except the first two, the best path length was found during or immediately after the first MCTS episode which again indicates that the Q-function is sufficient for finding assembly sequences with short removal paths.

VIII. Discussion and Future Work

We demonstrated an approach for optimizing RASP with a focus on minimizing assembly path lengths. By combining MCTS with deep Q-learning – where we used GNN to learn the Q-function – we could substantially enhance planning efficiency. We discussed a principled method for generating complex 3D assemblies from aluminium profiles and used it to create two datasets containing 14 assemblies with 21 removable parts and 7 assemblies with 30 removable parts, respectively. When training and testing on the same dataset our approach could outperform a vanilla MCTS. We could also successfully transfer knowledge from one dataset to the other. In the future, we aim to extend our methodology to more complex assemblies requiring multiple robots and/or tools.

References

[1] K.-L. Lee, A. Roesinger, and U. Hommel, "Development and practice of industrie 4.0 in china—practical experience of a german industrial software company in china," *Sci*, vol. 4, no. 3, 2022.

[2] B. Deepak *et al.*, "Assembly sequence planning using soft computing methods: A review," *Proceedings of the Institution of Mechanical Engineers, Part E: Journal of Process Mechanical Engineering*, vol. 233, 2019.

[3] L. S. H. de Mello, "Task sequence planning for robotic assembly," Ph.D. dissertation, Carnegie Mellon University, 1989.

[4] R. Andre and U. Thomas, "Anytime assembly sequence planning," in *Proceedings of ISR 2016: 47st International Symposium on Robotics*. VDE, 2016.

[5] T. Ebinger *et al.*, "A general and flexible search framework for disassembly planning," in *2018 IEEE International Conference on Robotics and Automation (ICRA)*, 2018.

[6] J. Wang, J. Liu, and Y. Zhong, "A novel ant colony algorithm for assembly sequence planning," *The International Journal of Advanced Manufacturing Technology*, vol. 25, 2005.

[7] J. Yu and C. Wang, "A max–min ant colony system for assembly sequence planning," *The International Journal of Advanced Manufacturing Technology*, vol. 67, 2013.

[8] H. Lv and C. Lu, "A discrete particle swarm optimization algorithm for assembly sequence planning," in *2009 8th International Conference on Reliability, Maintainability and Safety*, 2009.

[9] T. Kiyokawa, J. Takamatsu, and T. Ogasawara, "Assembly sequences based on multiple criteria against products with deformable parts," in *2021 IEEE International Conference on Robotics and Automation (ICRA)*, 2021.

[10] T. Kiyokawa *et al.*, "Many-objective-optimized semi-automated robotic disassembly sequences," *arXiv:2401.01817*, 2024.

[11] L. Ma *et al.*, "Planning assembly sequence with graph transformer," *2023 IEEE International Conference on Robotics and Automation (ICRA)*, 2023.

[12] M. Atad, J. Feng, I. Rodríguez, M. Durner, and R. Triebel, "Efficient and feasible robotic assembly sequence planning via graph representation learning," *2023 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2023.

[13] A. Cebulla, T. Asfour, and T. Kröger, "Speeding up assembly sequence planning through learning removability probabilities," in *2023 IEEE International Conference on Robotics and Automation (ICRA)*, 2023.

[14] J. Schrittwieser *et al.*, "Mastering atari, go, chess and shogi by planning with a learned model," *Nature*, vol. 588, no. 7839, 2020.

[15] N. Funk, G. Chalvatzaki, B. Belousov, and J. Peters, "Learn2assemble with structured representations and search for robotic architectural construction," in *Conference on Robot Learning*, 2021.

[16] A. Cebulla, T. Asfour, and T. 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)*, 2023.

[17] R. Andre and U. 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.

[18] I. Rodríguez *et al.*, "Pattern recognition for knowledge transfer in robotic assembly sequence planning," *IEEE Robotics and Automation Letters*, vol. 5, no. 2, 2020.

[19] J. Collins, M. Robson, J. Yamada, M. Sridharan, K. Janik, and I. Posner, "Ramp: A benchmark for evaluating robotic assembly manipulation and planning," 2023.

[20] B. Sundaralingam *et al.*, "curobo: Parallelized collision-free minimum-jerk robot motion generation," 2023.

[21] L. Kocsis and C. Szepesvari, "Bandit based monte-carlo planning," in *European Conference on Machine Learning*, 2006.

[22] C. B. Browne *et al.*, "A survey of monte carlo tree search methods," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 4, no. 1, 2012.

[23] T. Vodopivec, S. Samothrakis, and B. Ster, "On monte carlo tree search and reinforcement learning," *J. Artif. Intell. Res.*, vol. 60, 2017.

[24] V. Mnih *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, 2015.

[25] P. Battaglia *et al.*, "Relational inductive biases, deep learning, and graph networks," *arXiv*, 2018.

[26] G. Li, C. Xiong, A. K. Thabet, and B. Ghanem, "Deepergcn: All you need to train deeper gcns," *arXiv:2006.07739*, 2020.