Efficient Multi-Objective Assembly Sequence Planning via Knowledge Transfer between Similar Assemblies

Alexander Cebulla, Tamim Asfour, and Torsten Kröger

***Abstract*— Industry 4.0 aims to automate the production of customized product variants, which presents several challenges, particularly in the realm of assembly sequence planning (ASP). Manufacturers are often interested not only in a viable sequence but also in optimizing multiple additional objectives. However, because there are N! potential sequences for an assembly containing N parts, discovering such sequences can be time-consuming. To accelerate this process, we propose an approach that combines Monte Carlo Tree Search (MCTS) with deep learning to effectively transfer knowledge between similar assemblies. Specifically, we employ learnable state-action functions using graph neural networks for two common objectives: minimizing the number of direction changes and maximizing part accessibility. After pretraining these functions on similar assemblies, we could use them to efficiently guide an MCTS such that it found assembly sequences that optimized both objectives for two sets of 3D puzzles consisting of either 38 or 58 parts. In fact, for both sets, our approach outperformed both the unmodified MCTS and an MCTS that utilized state-action functions trained during the search.**

I. Introduction

A goal of industry 4.0 is the automatic production of customized product variants. However, to achieve this, several challenges must be overcome. A significant one is assembly sequence planning (ASP), which involves determining the optimal sequence of assembly steps required to assemble a final product. One approach for automating ASP is assembly-by-disassembly (AbD), which, given a 3D model of a fully assembled product, finds an assembly sequence by first finding a disassembly sequence and then inverting it. This is done by iteratively testing in simulation, whether each part can be removed without collision until all parts are removed.

Each disassembly sequence must satisfy fundamental constraints, such as ensuring that each part can be removed without colliding with other parts or compromising the stability of the assembly. However, manufacturers often seek to optimize additional objectives alongside these constraints [1]. For instance, they may aim to minimize the number of direction changes in the sequence to simplify the mounting of parts or prioritize maximizing the accessibility of each part during the assembly process. Finding a sequence that optimize such objectives can be time-consuming, given that the search space for assembly sequences grows exponentially, with N! potential sequences for an assembly with N parts.

To accelerate this process, we use deep learning methods to guide a multi-objective Monte Carlo tree search (MCTS) [2]. This approach has shown promising results when dealing with large state spaces [3]. Recently, it has been successfully applied to ASP [4]. However, the focus was on optimizing a single objective. Our approach aims at generalizing these ideas, allowing to optimize multiple objectives at once, as shown in Fig. 1.

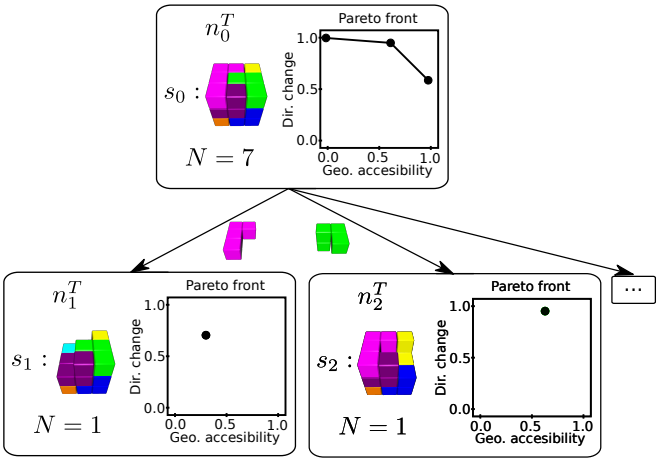


Fig. 1: Multi-objective assembly sequence planning for a soma cube using a variant of Monte Carlo Tree search [2].

Specifically, this knowledge has to be provided in the form of state-action functions, also known as Q-functions, which calculate an estimation of the maximum expected costs over all potential future sequences given the current state of an assembly and a part to be removed.

We discuss how such functions can be learned for two specific objectives – the number of direction changes and the accessibility of each part – using graph neural networks (GNNs). We evaluate our approach, by optimizing these two costs while assembling two sets of 3D puzzles consisting of either 38 or 58 parts. We can demonstrate that using pretrained Q-functions to guide an MCTS outperforms the unmodified MCTS, as well as an MCTS that utilized Q-functions that were trained during the search. In summary, our main contributions are:

1) an approach for multi-objective ASP that combines a multi-objective MCTS [2] with deep learning to effectively transfer knowledge from similar assemblies.
2) the implementation of learnable Q-functions for two specific objectives, namely the number of direction changes and the accessibility of each part, using GNNs.

The research leading to these results has received funding from the Carl Zeiss Foundation.

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

II. Related Work

Various approaches have been discussed over the years for automatically generating assembly sequences that are optimized with respect to various objectives (see [5] for an overview). In [6], assemblies were represented as AND/OR graphs, where each node represents a (sub-)assembly. They use two types of edges: Nodes connected by AND edges represent specific assembly sequences. OR edges, on the other hand, represent choices, indicating different assembly sequences that both contain a specific sub-assembly. Using this representation, assembly sequences were planned that were optimized with respect to three different objectives. This representation was also used in [7], where an anytime search algorithm was presented that produced assembly sequences optimized with respect to various objectives. Similarly to our work, distance maps [8] were used to define cost functions for the geometric accessibility of parts and the number of direction changes required to assemble each part. The geometric accessibility was also maximized in [9] via deep reinforcement learning. Another common approach to multi-objective optimization of assembly sequences is using a genetic algorithm [10], [11]. In [10], such an algorithm was employed to find an assembly sequence that minimizes both the number of direction changes and the assembly time. Similarly, in [11], the objective was to minimize not only the number of direction changes but also the number of tool changes. However, these approaches do not utilize transfer learning and require starting the planning process from scratch. Conversely, strategies that rely on case-based planning (CBP) adopt a different approach by reusing assembly sequences of sub-assemblies from related objects. For instance, in [12], a database was used to retrieve assembly plans for sub-assemblies based on similarities in part connections, mating directions, and mating constraints. Another example is [13], where a genetic algorithm inferred an assembly sequence for a new assembly based on correspondences between its parts and a reference assembly. Furthermore, CBP was used in [14] to generate feasible assembly sequences for a new product by reusing assembly sequences of old products within the same product family. Recent approaches [15], [4], [16] rely on graph-based representations of assemblies in combination with deep learning approaches to utilize previously learned knowledge. In [15], a GNN was utilized to estimate the probability of successfully removing assembly parts during the AbD process. By prioritizing the testing of parts with a higher predicted removability probability, a reduction in the number of failed removal attempts ranging from 64% to 90% was achieved, depending on the specific product being tested. Similarly to our approach [4] used trained Q-functions in combination with MCTS to enable autonomous robotic assembly of complex 3D structures. In [16], it was demonstrated that large-scale reinforcement learning is sufficient to train agents that can assemble complex unseen blueprints. However, these works focus on finding feasible sequences that are collision-free and stable. In particular, [4], [16] optimized a single objective that linearly combines a stability cost with a cost that measures how close a part could be placed to its target as given by a blueprint. In contrast, our approach allows manufacturers to optimize multiple costs, such as the number of direction changes or geometric accessibility, and select a solution that best reflects their desired trade-offs.

III. Problem Statement

Let an assembly be a set of its parts $\mathcal{A} := \{p_1, p_2, \dots p_N\}$, where each part p_i is defined by its geometric shape and 6D pose. Further, let $\boldsymbol{x} \in X$ be a decision vector that encodes a disassembly sequence s.t. its first element $\boldsymbol{x}_0 := j; p_j \in \mathcal{A}$ corresponds to the index of the first part that was removed from assembly $\mathcal{A}$. X is the set of all viable sequences, i.e., sequences where every part can be removed without collision and where the assembly remains stable during all removal steps. We now define the vector function $\boldsymbol{f}(\boldsymbol{x}) = (f_1(\boldsymbol{x}), f_2(\boldsymbol{x}), \dots, f_m(\boldsymbol{x}))$ that consists of $m \geq 2$ possibly objective functions $f_i : \mathcal{R}^N \to \mathcal{R}, i \in \{0, \dots, m\}$.

The problem is then given by

$$\max_{\boldsymbol{x} \in X} \boldsymbol{f}(\boldsymbol{x})$$

There may be multiple solutions that optimize different objectives. Given two potential solutions $\boldsymbol{x}, \boldsymbol{y} \in X$, $\boldsymbol{x}$ dominates $\boldsymbol{y}$, denoted by $\boldsymbol{x} \prec \boldsymbol{y}$, if $f_i(\boldsymbol{x}) \geq f_i(\boldsymbol{y})$ for all $i = 1, 2, \dots, m$ and there exists at least one k such that $f_k(\boldsymbol{x}) > f_k(\boldsymbol{y})$. The goal is to find a set of non-dominated solutions $\mathcal{P}$ – i.e., no solution dominates any other solution in the set – that maximizes all objectives simultaneously. In fact, if for all solutions $x \in P$ there exists no other solution $y \in X$ that dominates it, then $\mathcal{P}$ is called Pareto set, and the corresponding objective vectors form the Pareto front $\mathcal{P}^f := \boldsymbol{f}(\boldsymbol{x}), \forall \boldsymbol{x} \in \mathcal{P}$.

IV. Multi-objective Optimization for Assembly Sequence Planning

We utilize the AbD approach to find assembly sequences that are optimized with respect to multiple objectives. The general idea of the AbD approach is to find viable disassembly sequences for a product which can then be inverted to obtain viable assembly sequences. However, obtaining and evaluating these sequences is challenging, as the full evaluation of all possible disassembly sequences suffers from a combinatorial explosion: $N!$ potential sequences would have to be evaluated for an assembly with N parts.

To address this issue and to obtain the Pareto set of assembly sequences defined in Sec. III more efficiently, we propose to guide a multi-objective MCTS [2] with learnable Q-functions. These functions are trained to estimate the reward of actions used in the disassembly process (like removing a part), based on the current disassembly state (i.e. without actually performing actions like removing parts). They allow us to construct the search tree created by the MCTS more efficiently and to find optimal assembly sequences faster/with fewer steps. To learn the Q-functions, we train GNNs on a graph-encoding of the disassembly state.

In the following, we first formalize ASP as a Markov decision process with multiple rewards. We then describe how we used Q-functions to guide a multi-objective MCTS [2]. Next, we introduce the fundamental constraints that must be satisfied by each disassembly step, along with the reward functions for the objectives we aim to optimize. We then illustrate how we developed learnable Q-functions for these objectives: First, we present how we encoded all required state information as graph, then we discuss how we used this encoding to train GNNs as Q-functions.

A. Assembly Sequence Planning as Markov Decision Process

A Markov Decision Process (MDP) is a tuple (S, A, P, R), where S is the set of states and A is the set of actions. Then, $P(s_t, a_t, s_{t+1})$ is the probability of transitioning from state s_t to state s_{t+1} under action a_t, and $R(s_t, a_t, s_{t+1})$ represents the immediate reward obtained during the transition. Given an MDP, the goal is to find an optimal policy $\pi : S \to A$, which is a function mapping states to actions, such that the expected reward is maximized:

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

where $0 \leq \gamma \leq 1$ a discount factor.

One approach to finding an optimal policy π is deep Q-learning (DQL) [17]. A Q-function, also known as state-action function, estimates the expected discounted reward for taking a specific action a in a given state s then following a particular policy π thereafter. The goal of DQL is to first train a neural network to approximate the Q-function for an optimal policy. 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 maintains a replay buffer that stores experiences $(s_t, a_t, s_{t+1}, R(s_t, a_t, s_{t+1}))$ gained while traversing the MDP. Then, during the training process, it samples batches of experiences from the replay buffer to update the neural network's parameters. This allows the algorithm to break correlations between sequential experiences and learn more efficiently from a diverse set of data points.

ASP by AbD can be modeled as a finite MDP, where at each step, it must be decided which part needs to be removed next. We define S to be the set of all disassembly states $s \subseteq \mathcal{A}$, where an disassembly state contains all parts that are still present in the assembly. Further, we define an action as removing a part. Specifically, if at time step t part $p_i \in \mathcal{A}$ was removed, we define the corresponding action to be equal to the part's index, i.e., $a_t = i$. Given that the disassembly process is deterministic, $P(s_t, a_t, s_{t+1}) = 1$ for all viable transitions – that is, part a can be removed in state s_t without collision and the disassembly state s_{t+1} is stable with respect to gravity – and 0, otherwise. Finally, because we want to optimize multiple objectives, we define $R(s_t, a_t, s_{t+1}) := (R_1(s_t, a_t, s_{t+1}), R_2(s_t, a_t, s_{t+1}), ..., R_m(s_t, a_t, s_{t+1}))$; $m \geq 2$ to be a vector reward function, where $R_i(s_t, a_t, s_{t+1})$ is the i-th reward obtained when transitioning from state s_t to state s_{t+1} under action a.

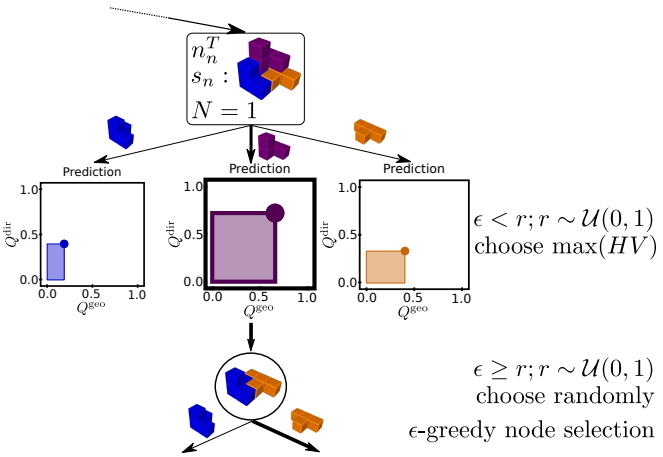


Fig. 2: Use of ϵ-greedy policy during MCTS simulation step. The hypervolume is represented by the shaded areas.

B. Guided Multi-objective Monte Carlo Tree Search for Assembly Sequence Planning

We applied a variant of MCTS [2] to search for assembly sequences that optimize various objectives. In detail, this algorithm builds a search tree as depicted in Fig. 1) through repeating the following four steps until a search budget is exhausted: selection, expansion, simulation, and backpropagation. Each tree node n^T stores a disassembly state $s \in S$. Additionally, each node maintains a local approximation of the Pareto front $\mathcal{P}^f$ and tracks the number of visits to that specific node, denoted as N. Both are updated during the backpropagation step.

The children of each node represent all viable disassembly states reachable from the parent node's disassembly state. Formally, $n'^T \in \text{Children}(n^T) \iff \exists a \in A : P(s, a, s') = 1$, where s' and s are the disassembly states stored in nodes n'^T and n^T, respectively. A is the set of actions, and $P(s, a, s')$ represents the transition probability, as defined in Sec. IV-A.

We now provide a detailed explanation of the steps:

a) Selection: Starting from the root node n_0^T, child nodes are recursively selected using a selection policy until a non-expanded node is reached. That is, the node has at least one child node that has not been visited.

In single-objective MCTS, commonly the upper confidence bound (UCB) [18] is used as selection policy:

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

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

where n'^T is a child of n^T. In this case, instead of the Pareto front only a single value Q^T is stored in the tree nodes, which is the total reward obtained after visiting the corresponding node.

A variant of the UCB, that can handle Pareto sets, was proposed in [2]. It utilized the hypervolume indicator [19]. This indicator is commonly used to evaluate the quality of a set of solutions by measuring the size of the dominated portion of the objective space. Formally, the hypervolume $HV(\mathcal{P}, z)$ of $\mathcal{P}$ with respect to a reference point $z \in \mathbb{R}^m$ is given by:

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

where Λ is the Lebesgue measure. Using this metric, multi-objective UCB (MOUCB) can be defined as [2]:

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

b) Expansion: One of the unvisited child nodes is added to the tree.

c) Simulation: Starting from the added child, a policy is used to recursively select child nodes until a leaf node is reached. Nodes visited during this step are not added to the tree. It is in this step, that we utilize the trained Q-functions to guide the MCTS exploration, as illustrated in Fig. 2. In detail, we use an ϵ-greedy policy to facilitate exploration: if a random sample $r \sim \mathcal{U}(0, 1)$ is larger than a predefined threshold ϵ, we use the trained Q-functions to estimate the values of all viable actions with respect to their corresponding objectives. For each action, all computed values are multiplied which corresponds to computing the hypervolume indicator for a solution set with a single element. These volumes are then normalized. Finally, the action with the highest value is selected and the associated part is removed.

d) Backpropagation: After the simulation step, a reward vector $\boldsymbol{r}$ is computed by summing over all obtained rewards. During its backpropagation through the expanded nodes, it is either dominated by one or more elements of the local Pareto front – in which case $\boldsymbol{r}$ is discarded and the backpropagation is halted – or it is not dominated. In the later case, $\boldsymbol{r}$ is added to the local Pareto front. Furthermore, all elements in the set that are dominated by $\boldsymbol{r}$ are removed.

Throughout each MCTS episode, the four steps are continuously executed until the search budget is exhausted. At this point, a child node is chosen using the MOUCB, and the search process starts again. This is repeated until a leaf node is reached, signaling the end of the MCTS episode. The next MCTS episode starts again from the root node n_0^T.

C. Fundamental Constraints and Objectives

One of the fundamental constraints for ASP is that each part must be removable along a collision-free path. Various methods have been proposed to compute such paths. One popular approach is the mating vector method, which assumes that each part can be removed along a pre-computed direction. $2\frac{1}{2}$D distance maps [8] offer an efficient way to organize these vectors.

Let $\mathcal{M}(p_i, p_j)$ be a distance map of size $N \times N$. Each entry $\mathcal{M}_{(k,l)}$ encodes a direction vector $\boldsymbol{m}$ and separation distance d_s, which describes how far the part p_i can be moved into the direction defined by $\boldsymbol{m}$ before either colliding with part p_j or being removed, i.e., a maximum removal distance $d_{\max}$ is exceeded. To encode the direction, a stereographic projection is used that maps the indices of a 2D grid to 3D directions.

Maps are pre-computed for each pair of parts $(p_i, p_j) \in \mathcal{A} \times \mathcal{A}$. By combining all maps for a part p_i it can be determined how far it can be moved with respect to all other parts and directions. Formally,

$$\bar{\mathcal{M}}(p_i, s) := \min_{p_j \in s \setminus p_i} \mathcal{M}(p_i, p_j), \tag{1}$$

where $s \in S$ is a disassembly state as defined in Sec. IV-A. Thus, $\bar{\mathcal{M}}_{k,l}(p_i, s)$ is the shortest distance that p_i can be moved along a separation direction encoded in the distance maps before it collides with any other part in the disassembly state s.

a) Collision-free Removal Constraint: To determine whether a part p_i can be safely removed, one can check if there exists a direction $\boldsymbol{m}$ for which the minimum separation distance d_s with respect to all other parts present in the disassembly state s exceeds the maximum removal distance $d_{\max}$. Specifically, p_i can be safely removed, if there exist $k, l \in \{0, \ldots, N\}$ such that $\bar{\mathcal{M}}_{k,l}(p_i, s)$ is greater than $d_{\max}$.

b) Gravity Constraint: To ensure that every disassembly state is stable, all parts in the state must be supported. After each disassembly action, we use $\bar{\mathcal{M}}$ to verify that the shortest distance each part can be moved along the gravity vector's direction is zero.

c) Geometric Accessibility Objective: Having multiple options for attaching a part allows for greater flexibility during the assembly process and can also reduce the likelihood of damaging parts. To approximate how accessible a part p_i is in the current disassembly state s, one can sum over how far it can be moved along each direction before colliding with any other part present in the disassembly state s. Formally, we define the reward function for the geometric accessibility objective $R_{\text{geo}}(p_i, s)$ for part p_i in disassembly state s as:

$$R^{\text{geo}}(p_i, s) = \sum_{k,l \in 0,\ldots,N} \bar{\mathcal{M}}_{k,l}(p_i, s), \tag{2}$$

where $\bar{\mathcal{M}}_{k,l}(p_i, s)$ is the minimum over all distance maps for part p_i as defined in Eq. 1.

d) Direction Change Objective: Reducing the number of direction changes needed during an assembly process improves its overall efficiency and speed as fewer movements are required to assemble the parts. We optimize this by maximizing the number of directions in which each part can be removed, while taking into account the directions along which previous parts were removed.

Assume that in disassembly state s_n disassembly action a_n is executed, which will remove part p_i. Additionally, a sequence of previously visited disassembly states required to reach s_n as well as the corresponding disassembly actions (i.e., parts that were removed) in each state $\mathcal{D}_n = ((s_0, a_0), (s_1, a_1), \ldots, (s_{n-1}, a_{n-1}))$ is given. Also, let $B^{\bar{\mathcal{M}}}(p_i, s)$ be a binary map computed from $\bar{\mathcal{M}}_{k,l}(p_i, s)$ which is the minimum over all distance maps for part p_i

as defined in Eq. 1. It is one for all possible directions in which a part p_i can be removed in disassembly state s, and zero, otherwise. Then, we define a function $H(a_n, s_n, \mathcal{D}_n)$ that computes the directions in which part p_i can be removed in state s_n that are in common with the potential directions along which the previous parts in the sequence $\mathcal{D}_n$ were removed:

$$H(a_n, s_n, \mathcal{D}_n) = B^{\bar{\mathcal{M}}}(p_i, s) \cap H(a_{i-1}, s_{n-1}, \mathcal{D}_{n-1}) \tag{3}$$

If there are no common directions with the previous parts in $\mathcal{D}_n$, we assume a direction change and restart the computation with all the directions in which part p_i can be removed in state s_n:

$$H(a_n, s_n, \mathcal{D}_n) = B^{\bar{\mathcal{M}}}(p_i, s)$$

For the first action a_0, we use all the directions in which its corresponding part could be removed:

$$H(a_0, s_0, \mathcal{D}_0) = B^{\bar{\mathcal{M}}}(p_k, s_0),$$

where we assume that action a_0 removed part p_k in the first disassembly state s_0 that corresponds to the full assembly $\mathcal{A}$. $\mathcal{D}_0 = ()$ is always an empty sequence.

The reward function for the direction change objective is then defined as:

$$R^{\text{dir}}(p_i, s_n, \mathcal{D}_n) = \sum_{k,l \in 0,\ldots,N} H_{k,l}(a_n, s_n, \mathcal{D}_n) \tag{4}$$

That is, the sum over all the directions in which part p_i can be removed in state s_n that are in common with the potential directions along which the previous parts in the sequence $\mathcal{D}_n$ were removed. However, in case of a direction change, we do not provide any reward. Formally, if $B^{\bar{\mathcal{M}}}(p_i, s) \cap H(a_{i-1}, s_{n-1}, \mathcal{D}_{n-1}) = \emptyset$:

$$R^{\text{dir}}(p_i, s_n, \mathcal{D}_n) = 0 \tag{5}$$

D. Encoding Assemblies as Graph

A disassembly state $s \in S$ is encoded as a directed graph $G_s = (N_s, E_s)$ where N_s is the node set and E_s is the edge set. Each part p_i in s is mapped to a node $n_i \in N_s$, and connections between parts p_i and p_j are captured by directed edges $e_{i,j} \in E_s$. Finally, each node n_i has a node attribute $\boldsymbol{u}_{n_i}$ and each edge $e_{i,j}$ has an edge attribute $\boldsymbol{v}_{e_{i,j}}$.

To create a graph encoding G_s^{geo} of a disassembly state s that can be used to learn a Q-function for the geometric accessibility reward function defined in Eq. 2, we use the distance maps $\mathcal{M}(p_i, p_j)$ for parts p_i and p_j as edge attributes for the corresponding directed edges. Specifically, edge attribute $\boldsymbol{v}_{e_{i,j}}$ is set to $\mathcal{M}(p_i, p_j)$, and for $\boldsymbol{v}_{e_{j,i}}$, it is set to $\mathcal{M}(p_j, p_i)$. Additionally, the node attribute for part p_i is given by $\boldsymbol{u}_{n_i} := \bar{\mathcal{M}}(p_i, s)$.

The graph encoding for the direction change reward function $G^{\text{dir}}_{(s,\mathcal{D})}$ defined in Eq. 4 and 5, depends not only on the current disassembly state s, but also on the sequence $\mathcal{D}$ of previously visited disassembly states and actions required to reach s.

The encoding of current disassembly state s is similar to its encoding for the geometric accessibility reward function. The only difference is that we use the binary version of the distance map $B^{\mathcal{M}}(p_i, p_j)$. That is, it is one for all possible directions in which a part p_i can be removed in the presence of part p_j, and zero, otherwise. Then, the edge attribute $\boldsymbol{v}_{e_{i,j}}$ is set to $B^{\mathcal{M}}(p_i, p_j)$, and for $\boldsymbol{v}_{e_{j,i}}$, it is set to $B^{\mathcal{M}}(p_j, p_i)$. Additionally, the node attribute for part p_i is given by $\boldsymbol{u}_{n_i} := B^{\bar{\mathcal{M}}}(p_i, s)$.

With regards to $\mathcal{D}$, we need to encode the common directions along which the previous parts were removed. This is computed by function $H(a_{n-1}, s_{n-1}, \mathcal{D}_{n-1})$ that is defined in Eq. 3. Given that the output of this function is itself a binary map, we can directly encode it as the node attribute $\boldsymbol{u}_{n_c} := H(a_{n-1}, s_{n-1}, \mathcal{D}_{n-1})$ of an additional node n_c that we add to the graph $G^{\text{dir}}_{s,\mathcal{D}}$. We connect n_c with outgoing edges to all other nodes.

A limitation of this particular encoding method is that it produces a fully-connected, directed graph. That is, for a disassembly state s consisting of N parts, i.e., $|s| = N$, the number of edges grows quadratically: $|E_s| = N^2 - N$. To reduce the number of edges, for each part $p_i \in s$, we use the corresponding distance maps to determine the N_c parts closest to it and then only added edges to these parts.

E. Using GNN as learnable Q-functions

GNNs [20] consist of multiple layers where each layer l takes a graph $G^l = (N^l, E^l)$ as input, performs several transformations, and outputs another graph $G^{(l+1)} = (N^{(l+1)}, E^{(l+1)})$ with updated node and edge attributes of the same shape. Let Φ^l be the l-th layer of a GNN, then

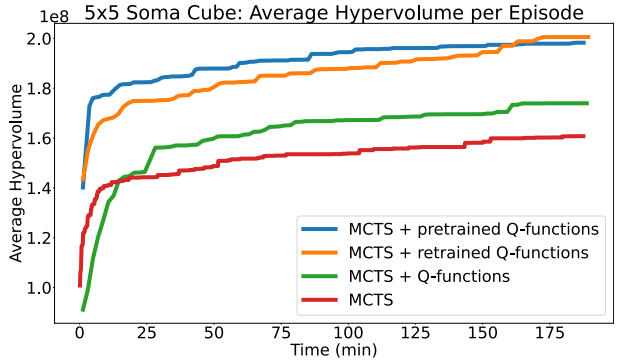
$$G^{(l+1)} = \Phi^l\left(G^l\right), \forall l \in [0, L-1],$$

where L the total number of layers. To update an edge attributes a feed-forward neural network (FNN) is used that takes the edge and the adjacent node attributes as input. Then the node attributes are updated by first aggregating attributes from adjacent nodes and edges and then providing them – together with the node's attributes – as input to another FNN. Formally, for the l-th layer of the GNN, let ϕ_e^l and ϕ_n^l be the FNNs used to update the node attributes $\boldsymbol{u}_{n_i}^{(l+1)}$ and edge attributes $\boldsymbol{v}_{e_{i,j}}^l$ computed by the previous layer, then:

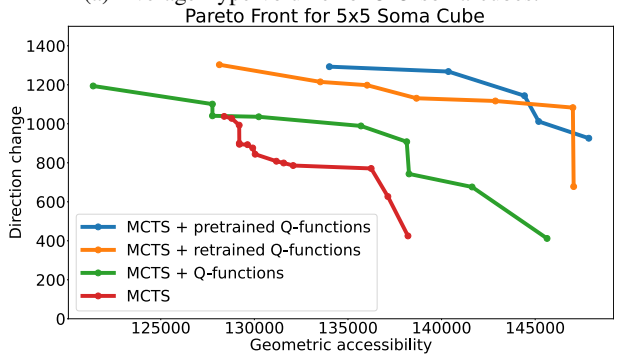
$$\begin{aligned}
\boldsymbol{v}_{e_{i,j}}^{(l+1)} &= \phi_e^l(\boldsymbol{u}_{n_i}^l + \boldsymbol{v}_{e_{i,j}}^l + \boldsymbol{u}_{n_j}^l) \\
\bar{\boldsymbol{u}}_{n_i}^{(l+1)} &= \min\left(\{\text{ReLU}\left(\boldsymbol{u}_{n_i}^l + \boldsymbol{v}_{e_{i,j}}^{(l+1)}\right) + \epsilon | e_{i,j} \in E_s\}\right) \\
\boldsymbol{u}_{n_i}^{(l+1)} &= \phi_n^l\left(\boldsymbol{u}_{n_i}^l + \bar{\boldsymbol{u}}_{n_i}^{(l+1)}\right),
\end{aligned}$$

We now describe how we used a GNN to approximate the Q-function Q^{geo} for the geometric accessibility reward function based on the graph embedding G_s^{geo} described in Sec. IV-D. Let $G_s^{L,\text{geo}}$ be the graph computed by the GNN, with the corresponding node set $N_s^{L,\text{geo}}$. The Q-function is defined as

$$Q^{\text{geo}}(s_t, a_t) = \phi_{\text{geo}}\left(\left[n_{a_t}^L, \frac{1}{|N_{s_t}^{L,\text{geo}}|} \sum_{n_j^L \in N_{s_t}^{L,\text{geo}}} n_j^L\right]\right),$$

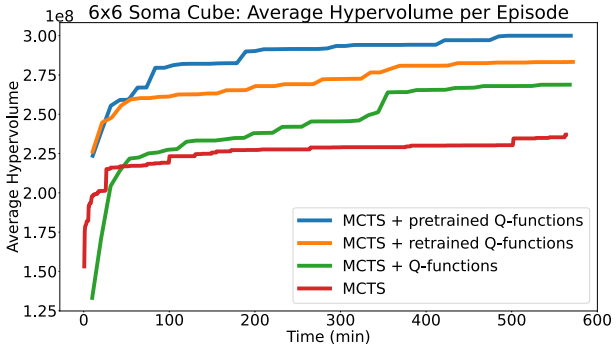


(a) Average hypervolume for 5x5 soma cubes.

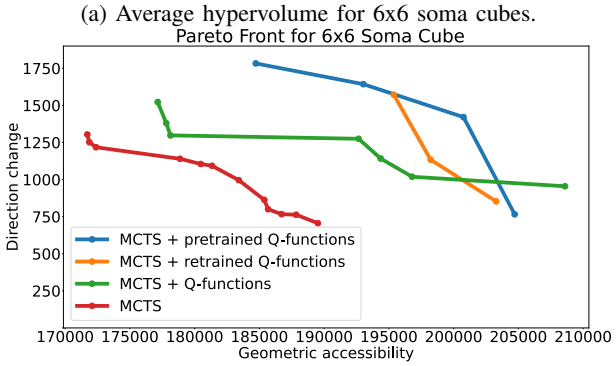


(b) Pareto front for a 5x5 soma cube.

Fig. 3: Average hypervolume and pareto front for 5x5 soma cubes.



(a) Average hypervolume for 6x6 soma cubes.



(b) Pareto front for a 6x6 soma cube.

Fig. 4: Average hypervolume and pareto front for 6x6 soma cubes.

where ϕ_{geo} is an FNN and $n^L_{a_t} \in N^{L,\text{geo}}_{s_t}$ represents the node embedding of the node that corresponds to the part removed by action a_t. We concatenate $n^L_{a_t}$ with the global mean over all node embeddings $N^{L,\text{geo}}_{s_t}$ and use it as input for an FNN ϕ_{geo}. The Q-function Q^{dir} for the direction change reward function is similarly defined.

V. Experiments

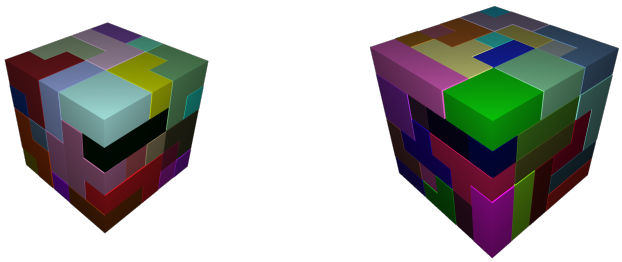
The goal of our experiments was to compare the performance of the unmodified multi-objective MCTS [2] against our proposed multi-objective MCTS guided via learned Q-functions. In particular, we tested four scenarios:

1) **MCTS**: This is the unmodified multi-objective MCTS.
2) **MCTS + Q-functions**: The Q-functions were trained after each MCTS episode and then used to guide the next episode of MCTS.
3) **MCTS + pretrained Q-functions**: The Q-functions were pretrained on previously obtained data.
4) **MCTS + retrained Q-functions**: A combination of the previous two approaches: the Q-functions were trained on previously obtained data and then retrained after each MCTS episode.

A search budget of 5 was used in all scenarios.

We conducted multiple experiments using two datasets of soma cubes (see Fig. 5). The first set consisted of eight cubes with size 5×5 and the second set of four cubes with size 6×6.

All Q-functions were trained via DQL [17] using a replay buffer of size 10000. After each MCTS episode, we add all



(a) 5×5 Soma cube (38 parts).

(b) 6×6 Soma cube (58 parts).

Fig. 5: Two soma cubes from the datasets.

obtained data – that is, data obtained during the selection and the simulation steps – to the replay buffer. We then sample a batch consisting of either one element, for the 6×6 soma cubes, or five elements for the 5×5 soma cubes and train the GNNs. For both cube sizes, we want to train on 100 samples, therefore, we used 20 training episodes for the 5×5 soma cubes and 100 training episodes for the 6×6 soma cubes. For experiments, where we pretrained the Q-functions, we used a two-fold cross validation for both datasets.

We used Python and relied on the Open3D [21] library for handling meshes, the Flexible Collision Library [22] to test for part collisions, and on PyTorch Geometric [23] for implementing GNNs. All experiments were performed on Ubuntu with an Intel Core i9-10980XE, 128 GB of memory, and a GeForce RTX 2080 Ti.

A. Results

Our experimental results for the 5×5 and 6×6 soma cube datasets are depicted in Fig. 3 and Fig. 4, respectively. The hypervolume on the y-axis in Fig. 3a and Fig. 4a was computed from the Pareto front stored in the root node of the Monte-Carlo tree at each corresponding timestep. These values were then averaged over either all eight or four cubes. Additionally, Fig. 3b and Fig. 4b show the final Pareto fronts for a soma cube from each dataset.

In all experiments, a consistent run time was maintained for each dataset. Specifically, we set the time limit based on the duration required to run either 50 Monte-Carlo tree search (MCTS) episodes for the 6×6 soma cube or 100 MCTS episodes for the 5×5 soma cube in the MCTS + Q-function setting. Once this threshold was reached, the search in other settings was halted. On average, the total running time was 190 minutes for the 5×5 soma cubes and 570 minutes for the 6×6 soma cubes.

Fig. 3a demonstrates that using Q-functions trained with data collected during the MCTS (MCTS + Q-functions) outperformed the vanilla MCTS approach for both the 5×5 and 6×6 soma cubes. It is noteworthy that during this period, the vanilla MCTS performed, on average, 14.4 times more episodes for the 5×5 soma cubes and 15 times more episodes for the 6×6 soma cubes. For both datasets, employing pretrained Q-functions (MCTS + pretrained Q-functions) yielded better results than the vanilla MCTS and the MCTS where Q-functions were trained during the search. Finally, retraining the pretrained Q-functions during the MCTS (MCTS + retrained Q-functions) did not lead to an improvement over solely utilizing pretrained Q-functions.

VI. Discussion and Future Work

We proposed an efficient approach for multi-objective ASP by leveraging learnable Q-functions for knowledge transfer among similar assemblies. To evaluate this approach, we introduced graph-based encodings for two common objectives, namely, geometric accessibility and direction change, and used them to train two GNNs. We conducted experiments on two datasets of soma cubes and compared the performance for different settings. The results demonstrate that utilizing pretrained Q-functions outperformed both the vanilla MCTS and the MCTS with Q-functions trained during the search. For future work, we plan to extend our approach to more complex assemblies that require the planning of more intricate removal paths. To achieve this, we intend to incorporate metrics such as path smoothness and/or curvature that can be applied to such paths. This will allow us to evaluate our approach on a wider range of assemblies.

References

[1] R. Jones, R. Wilson, and T. Calton, "On constraints in assembly planning," *IEEE Transactions on Robotics and Automation*, vol. 14, no. 6, pp. 849–863, 1998.

[2] D. Perez, S. Mostaghim, S. Samothrakis, and S. M. Lucas, "Multiobjective monte carlo tree search for real-time games," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 7, no. 4, pp. 347–360, 2015.

[3] D. Silver and et. al, "Mastering the game of Go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, pp. 484–489, Jan. 2016.

[4] 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.

[5] B. Deepak, G. B. Murali, M. V. A. R. Bahubalendruni, and B. B. Biswal, "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, pp. 653 – 683, 2019.

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

[7] R. Andre and U. Thomas, "Anytime assembly sequence planning," in *Proceedings of ISR 2016: 47st International Symposium on Robotics*. VDE, 2016, pp. 1–8.

[8] 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, pp. 1–7.

[9] K. Kitz and U. Thomas, "Neural dynamic assembly sequence planning," *2021 IEEE 17th International Conference on Automation Science and Engineering (CASE)*, pp. 2063–2068, 2021.

[10] Y. K. Choi, D. M. Lee, and Y. B. Cho, "An approach to multi-criteria assembly sequence planning using genetic algorithms," *The International Journal of Advanced Manufacturing Technology*, vol. 42, pp. 180–188, 2009.

[11] Y.-J. Tseng, J.-Y. Chen, and F.-Y. Huang, "A multi-plant assembly sequence planning model with integrated assembly sequence planning and plant assignment using ga," *The International Journal of Advanced Manufacturing Technology*, vol. 48, pp. 333–345, 2010.

[12] A. Swaminathan and K. S. Barber, "Ape: an experience-based assembly sequence planner for mechanical assemblies," *Proceedings of 1995 IEEE International Conference on Robotics and Automation*, vol. 2, pp. 1278–1283 vol.2, 1995.

[13] Q. Su, "Applying case-based reasoning in assembly sequence planning," *International Journal of Production Research*, vol. 45, pp. 29 – 47, 2007.

[14] K. Lee, S. Joo, and H. I. Christensen, "An assembly sequence generation of a product family for robot programming," *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 1268–1274, 2016.

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

[16] S. K. S. Ghasemipour, D. Freeman, B. David, S. S. Gu, S. Kataoka, and I. Mordatch, "Blocks assemble! learning to assemble with large-scale structured reinforcement learning," in *International Conference on Machine Learning*, 2022.

[17] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. A. Riedmiller, A. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, pp. 529–533, 2015.

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

[19] E. Zitzler, "Evolutionary algorithms for multiobjective optimization: methods and applications," 1999.

[20] P. Battaglia, J. B. C. Hamrick, V. Bapst, A. Sanchez, V. Zambaldi, M. Malinowski, A. Tacchetti, D. Raposo, A. Santoro, R. Faulkner, C. Gulcehre, F. Song, A. Ballard, J. Gilmer, G. E. Dahl, A. Vaswani, K. Allen, C. Nash, V. J. Langston, C. Dyer, N. Heess, D. Wierstra, P. Kohli, M. Botvinick, O. Vinyals, Y. Li, and R. Pascanu, "Relational inductive biases, deep learning, and graph networks," *arXiv*, 2018. [Online]. Available: https://arxiv.org/pdf/1806.01261.pdf

[21] Q.-Y. Zhou, J. Park, and V. Koltun, "Open3D: A modern library for 3D data processing," *arXiv:1801.09847*, 2018.

[22] J. Pan, S. Chitta, and D. Manocha, "Fcl: A general purpose library for collision and proximity queries," in *2012 IEEE International Conference on Robotics and Automation*, 2012, pp. 3859–3866.

[23] M. Fey and J. E. Lenssen, "Fast graph representation learning with PyTorch Geometric," in *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.