Coordinated Multi-Robot Disassembly for Makespan Optimization of Large-Scale Assemblies

Niklas Hargus[1] and Andreas Orthey[1] and Marc Toussaint[1,2]

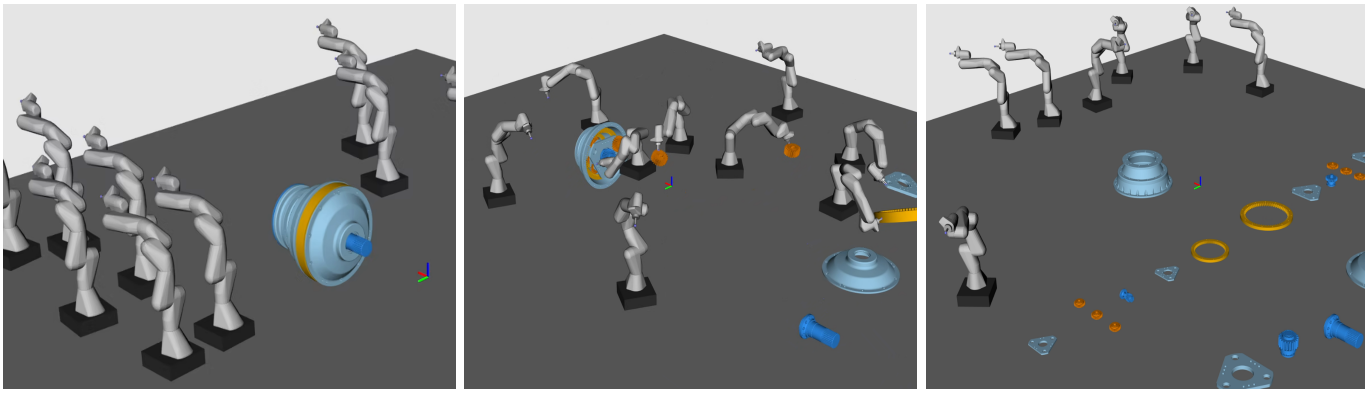


Fig. 1: We develop a coordinated disassembly multi-robot planner to solve large-scale multi-robot task and motion planning problems. Pictured is our framework coordinating 9 mobile manipulators which are disassembling a gearbox with 33 pieces. **Left:** Gearbox in assembled state, robots are approaching pick positions. **Middle:** Robots during disassembly with some robots transporting parts, while others are returning for the next pick. **Right:** Final disassembled state of gearbox with robots idle and object parts spread out on the floor.

***Abstract*—Multi-robot task and motion planning for disassembly tasks requires robots to operate in confined workspaces while coordinating their motions with other robots. To tackle this problem, we propose a planning method called coordinated multi-robot disassembly (CoMuDi). CoMuDi coordinates a team of robots for disassembly tasks. The input is a team of robots, an assembly of objects, and a dependency graph. Based on this information, we create compound tasks for pick, place, and exit motions. By propagating temporal constraints, we ensure that each robot can start and end their tasks as early as possible while avoiding collisions with nearby robots. By integrating the space-time RRT* planner (ST-RRT*) into CoMuDi, we ensure that individual tasks minimize arrival time and thereby help us minimize overall makespan. We compare the performance of CoMuDi using both ST-RRT* and RRT* planners with varying time bounds, demonstrating that the combination of CoMuDi and ST-RRT* leads to a higher success rate while minimizing makespan. Finally, we evaluate CoMuDi on six assemblies with up to 49 pieces and up to 9 robots. In those scenarios, we show that CoMuDi returns robot paths that exhibit low idle times, thereby demonstrating that CoMuDi can reliably solve large-scale assemblies. Videos can be found on https://sites.google.com/view/CoMuDi.**

I. Introduction

Coordinating multiple robots to disassemble objects is a fundamental skill for automating industries like recycling, repair, and remanufacturing. However, coordinating the motion and task assignments of robot teams is difficult. This is caused by robots needing to work in close proximity (e.g., multiple arms removing pieces from the same battery). Furthermore, the robots have to coordinate their tasks in time so that they minimize their idle time and decrease the total execution time.

To tackle those problems, we develop the coordinated multi-robot disassembly planner (CoMuDi). This algorithm combines existing ideas from multi-robot assembly [12] to coordinate task assignment, from scale-invariant sampling [2] to remove objects from tight, narrow passages, and from space-time planning [10] to minimize arrival time in individual tasks. This is supplemented by our main contributions, which involve a prioritized task queue to ensure that tasks are executed in the correct order and a temporal constraint propagation scheme to allow for more efficient coordination in time. The outcome of this planner are collision-free trajectories to disassemble arbitrary input objects, as is visualized in Fig. 1.

To summarize, our contributions include:

- We develop the coordinated multi-robot disassembly planner (CoMuDi), a prioritized sequential task and motion planning framework for multi-robot disassembly to minimize makespan. The framework features task queues to schedule and balance workloads, and a temporal constraint propagation mechanism to minimize idle time.
- We introduce a generalized disassembly task, which combines robot navigation, attach and detach tasks, object extraction and transport, and exit tasks into one cohesive task over which we minimize arrival time.
- We develop a collision checking framework that tracks attachment data and checks for future collisions to ensure that every robot and object is collision free.
- We implement the proposed framework in OMPL using ST-RRT* [10] as the underlying motion planner to handle unknown arrival times.

[1]Technical University of Berlin, Germany
[2]Robotics Institute Germany (RIG)

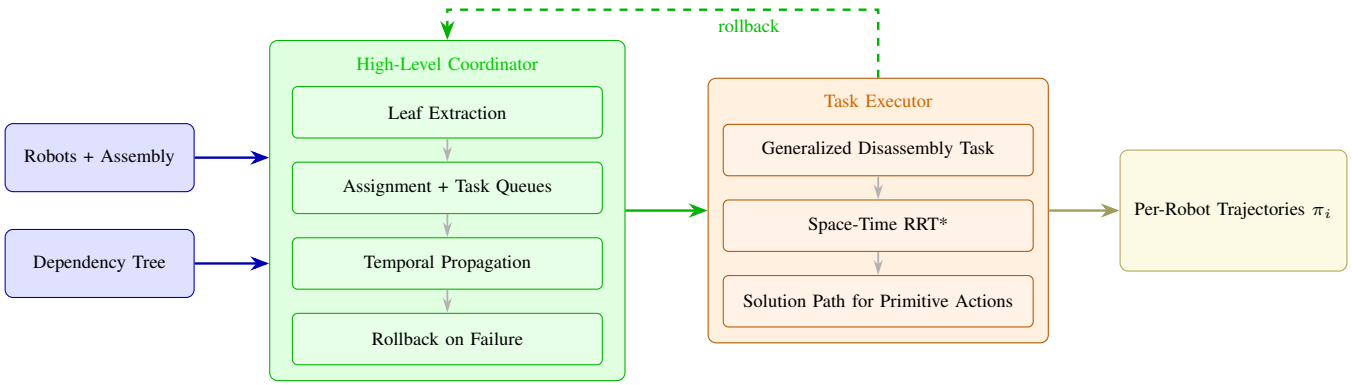


Fig. 2: Overview of the temporal multi-robot TAMP framework for disassembly tasks. As input, we require a team of robots and the assembly consisting of objects, together with a dependency tree specifying the possible execution order of the objects. This tree is used in the high-level coordinator to extract the next possible tasks by extracting the leaves.

II. Related Work

Our framework for multi-robot task and motion planning for disassembly problems builds upon multiple related approaches. This involves time-optimal motion planning with unknown arrival times, multi-robot navigation, task and motion planning, and disassembly sequence planning. We review those topics and discuss their relation to our work.

A. Time-Optimal Motion Planning

At the core of our approach lies the problem of time-optimal planning in a combined space-time space [14]. A particularly important problem hereby is to avoid dynamic obstacles with known obstacle trajectories. Approaches to tackle this problem involve roadmap-based dynamic planning approaches for arrival-time minimization [39, 40] or safe interval path planning [24], which represents the time dimension through intervals during which a configuration remains collision-free. The single-query tree alternative for space-time planning was first introduced by Sintov and Shapiro [29] with Time-Based RRT (TB-RRT) for rendezvous planning of two dynamic systems. However, it is limited by a fixed arrival time and lacks any optimality guarantees. Grothe et al.'s [10] extension to RRT* [16] treats time itself as a planning dimension. This allows RRTs to directly plan in space-time to find asymptotically optimal solutions with respect to arrival time. It grows a bidirectional tree from the start and finish at an unknown arrival time with a progressively widening time horizon. Our work is complementary to time-optimal motion planning as we rely on ST-RRT* [10] as an individual planner to optimize generalized disassembly tasks.

B. Multi-Robot Motion Planning

The two major approaches to multi-robot motion planning are separated by how the robots' individual configuration spaces are coupled. The centralized approach combines all configuration spaces together, which increases the complexity of the problem but remains optimal and complete, whereas the decoupled methods remain scalable, sacrificing completeness and optimality.

The discrete problem of Multi-Agent Path Finding (MAPF), stated by Yu and LaValle [43], for time- and distance-optimal MAPF on connected, undirected graphs, is NP-hard. The difficulties of the naive planning problem motivated Sharon et al. [26] to introduce Conflict-Based Search (CBS), a two stage decomposition into a high-level conflict-tree and low-level search in which conflicts are solved. CBS is optimal and complete on a grid-world and has grounded a large family of multi-agent path finders. The graph search approach of Phillips et al. [24] was extended by Andreychuk et al. [1] to multi-agent pathfinding with continuous time. This approach struggles to extend well to the high-dimensional configuration spaces of articulated manipulators.

Sampling-based multi-robot planning works in continuous, high-dimensional configuration spaces. Wagner and Choset [42] introduced M*, which searches an implicit representation of the combined roadmaps rather than constructing the joined roadmap, resulting in significant speedup. However, it is inefficient for higher dimensions due to the exponential growth of neighbors. This led to Solovey et al. [31] introducing discrete-RRT (dRRT), which extends M* with an oracle function to pick the best neighbor. Shome et al. [27] contributed the asymptotically optimal extension dRRT*, resulting in convergence to an optimum even in multi-robot manipulator spaces.

Erdmann and Lozano-Pérez [6] introduced prioritized planning to circumvent the complexity increase. They propose to assign an ordering to the robots and plan each one in turns against higher-priority robots, who are treated as dynamic obstacles in time. This approach trades completeness and optimality for gain in tractability.

Bennewitz et al. [3] demonstrated that fixed orderings can fail to find valid solutions, even though they exist. Čáp et al. [4] provides a characterization instance for which a revised prioritized algorithm is complete. Priority-Based Search (PBS) by Ma et al. [22] bridges prioritized planning and CBS by searching over partial orderings. However, these claims are only evaluated for mobile robots, rather than manipulators.

Our approach differs in that we focus on general task and motion planning, not solely on multi-robot navigation.

C. Task and Motion Planning

Task and Motion Planning (TAMP) addresses the problem in which a robot must simultaneously decide on what to do at a symbolic level and how to do it at a geometric level. The survey by Garrett et al. [8] distinguishes into three broad paradigms: classical hierarchical interface-based, sampling-based multi-modal, and optimization-based methods.

The classical TAMP formulation uses a symbolic planner to generate candidates. Kaelbling et al. [15] introduced an early work which provides a general strategy for hierarchical planning and execution by limiting the search space. Srivastava et al. [32] improves upon existing task and motion planning solutions by introducing a predicate-level interface between geometric and symbolic layer. Garrett et al. introduce PDDLStream [7], an extension to the Planning Domain Definition Language (PDDL) [9], which supports black-box implementations for grasps, placements and trajectories by providing adaptive algorithms that interleave symbolic search and constraint satisfaction.

Sampling-based TAMP uses mode switching in hybrid configuration spaces of intersecting sub-manifolds. Each mode switch amounts to sampling mode transitions and concatenating intra-mode paths. Siméon et al. [28] establish this perspective with their PRM-based manipulation planner, Hauser and Latombe [13] provide a more modern formalization with probabilistic completeness. Krontiris and Bekris [19] address non-monotone object rearrangement and Vega-Brown and Roy [41] provide the first asymptotically optimal guarantee with piecewise-analytic differential constraints for TAMP.

Toussaint [35] introduced Logic-Geometric Programming (LGP), in which a symbolic search over action sequences is interleaved with a non-linear program. This collapses the multi-level approach from previous approaches into a single optimization problem. KOMO [36] is used as the trajectory-level solver, a k-order Markov constrained NLP solved by Gauss-Newton with augmented Lagrangian. This integrated optimization-based approach was demonstrated to extend to contact dynamics and tool usage [38].

D. Multi-Robot Task and Motion Planning

Multi-robot task and motion planning (MR-TAMP) shares the same structure as single-robot TAMP with high-level decision making and low-level execution, but expands to task allocation, motion coordination and often makespan reduction.

Pan et al. [23] extended classic PDDL-based TAMP to multi-robot systems. They introduce a scheduler between the symbolic task planner and multi-robot motion planner. This scheduler is responsible for reducing the number of task planner calls and biasing the search based on geometric insights.

Toussaint and Lopes [37] generalized LGP to cooperative manipulation between a humanoid and Baxter robot but do not push to a larger team of robots. Hartmann et al. [12] demonstrates long-horizon multi-robot planning for construction work by decomposing large scale TAMP problems into smaller one robot subproblems. They leverage Logic-Geometric Programming [35] to solve for keyframes, which are then connected by ST-RRT* [10]. This was followed by formalizing the multi-arm multi-task planning problem based on greedy descent with random restarts to optimize the total makespan [11].

Chen et al. [5] approach multi-arm assembly problems by formulating a high-level mixed-integer program to solve for the task assignment under precedence constraints and a CBS-based low level planner to route the robots collision-free, yielding high-quality makespans at substantial computational cost.

Our work is complementary to those approaches in that we apply MR-TAMP to disassembly problems. While we also use ST-RRT* [10], similar to Hartmann et al. [12], our method includes scale-invariant sampling for integrating object extraction tasks. Additionally, we extend this to disassembly tasks by proposing a generalized disassembly task, the propagation of temporal constraints, and an efficient time-dependent collision-checking framework.

E. Disassembly Sequence Planning

Disassembly through robots connects the field of disassembly sequence planning (DSP) to robotics, by using the output of DSP as the input for robot task and motion planning. Lambert's survey [20] provides the classical foundation for DSP, which is formalized by graph- and AND/OR-based representations. Smith et al. [30] extend this representational line with Disassembly Sequence Structure Graphs (DSSG), which handle multi-target selective disassembly by encoding precedence and accessibility in a single structure.

The graph-based approach is contrasted by treating DSP as optimization-based problem. This line of work focuses on genetic algorithms (GA). Kongar et al. [18] and Lazzerini et al. [21] provide early work on GA-DSP, while Kheder et al. [17] provide extensions to address tool changes and accessibility violations under precedence constraints.

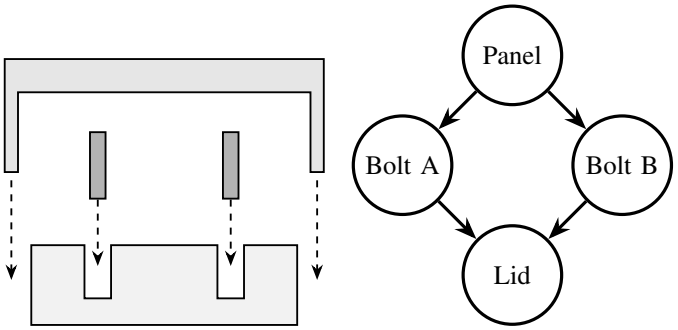
The precomputed nature of all DSP approaches sits in contrast to recent work by Poschmann et al. [25], which shifts to information-driven and learning-based approaches to acquire information at runtime about the assembly state.

III. Coordinated Multi-Robot Disassembly

A. Problem Statement

We consider a workspace $\mathcal{W}$ with N robots and K rigid objects. Each robot $r_i \in \mathcal{R} = \{r_1, \dots, r_N\}$ has a configuration space $\mathcal{Q}_i$ with d_i degrees of freedom. The compound state space is defined as $X = \mathcal{Q}_1 \times \cdots \times \mathcal{Q}_N$ with a combined dimension $d = \sum d_i$. Additionally, there exist K rigid objects $\mathcal{O} = \{o_1, \dots, o_K\}$ in $\mathcal{W}$, with each object $o_j \in \mathcal{O}$ being described by a starting pose $T^j_{\text{start}} \in SE(3)$ in the assembled state and a goal pose $T^j_{\text{goal}} \in SE(3)$ in the disassembled state.

a) Dependency Graph: Additionally, objects are not free to move in any sequence, but there exists a dependency graph G. G is a directed acyclic graph (DAG) that defines the order in which the objects have to be moved to accomplish the disassembly. G consists of vertices $\boldsymbol{V}$, representing the objects, and edges $\boldsymbol{E}$, representing the dependencies. As an example, Fig. 3 shows a dependency graph with four objects, whereby the panel depends on bolt A and B, which in turn



(a) Assembly **(b)** Dependency Graph

Fig. 3: Example of an assembly with a lid, two bolts, and panel. The dependency graph shows the assembly constraints.

depend on the lid. All leaf nodes in the graph have no dependencies on other nodes.

b) Assumptions: The goal of this paper is to coordinate a multi-robot team for disassembly. To study this in isolation, we make two assumptions to simplify the problem:

- The end-effector of each robot is modeled as an omnidirectional gripper, enabling grasps from any angle on the surface of the object. This abstracts away the problem of grasp planning, which differs depending on the gripper types.
- Execution of paths is done in a non-physical simulation. This ensures that effects such as stick-slip or object resting behaviors do not have to be considered.

This formulation allows us to concentrate on the task and motion planning problem itself. We believe those are reasonable assumptions that allow us to find solutions that act as necessary conditions for solving disassembly problems in the real world [34].

c) Goal: Given robot and object assembly, we define the *multi-robot disassembly problem* as the problem of finding a set of trajectories $\mathcal{T}$, with each trajectory $\pi_i \in \mathcal{T}$ being a sequence of pairs consisting of a configuration and an associated time point for robot r_i. They describe the collision free motion for each robot and objects to reach the disassembled state.

In this paper, we are also interested in the *optimal multi-robot disassembly problem*, which is the problem of finding a set of trajectories $\mathcal{T}^*$ that minimizes the makespan: the total time required for the disassembly process from start to finish.

B. Overview and Implementation

To solve the optimal multi-robot disassembly problem, we present the coordinated multi-robot disassembly (CoMuDi) planner. The pseudocode for CoMuDi is described in Alg. 1. The input to the algorithm is a precomputed dependency graph and all available robots. While the dependency graph is not empty (Line 1), the unassigned leaf nodes are obtained, which do not depend on any other vertex (Line 2). Next, all robots that currently do not have an assigned task are selected (Line 3). The algorithm then matches an available robot with a node based on an assignment policy (Line 6–7). It then creates a task in the robot's task queue (Line 8–9) and marks the node as assigned to avoid reassignment (Line 10), continuing until all possible pairings of leaf-robots have been exhausted (Line 5). Afterwards, it iterates through every task queue (Line 11), skipping any empty queues since no work needs to be done (Line 12–13). The first task in the queue is selected (Line 14), and the main run of the selected task is invoked (Line 15). If the task has failed (Line 16), the associated node is unmarked in the tree (Line 17) and is ready to be reassigned. If the task is successful, the constraints on the next node are propagated (Line 19), and the node is removed from the dependency tree (Line 20).

Algorithm 1: Coordinated Multi-Robot Disassembly

```
   Input: dGraph, robots
   Output: —
1. while dGraph ≠ ∅ do
2.     leaves ← getUnassignedLeaves(dGraph)
3.     freeRobots ← {r ∈ robots | taskQueue(r) = ∅}
4.     assignedTasks ← ∅
5.     while leaves ≠ ∅ ∧ freeRobots ≠ ∅ do
6.         robot ← pop(freeRobots)
7.         leaf ← pop(leaves)
8.         task ← createTask(robot, leaf)
9.         append(assignedTasks, task)
10.        markLeafAssigned(leaf)
11.    foreach (robot, queue) ∈ assignedTasks do
12.        if queue = ∅ then
13.            continue
14.        currentTask ← pop(queue)
15.        run(currentTask)
16.        if currentTask.failed then
17.            markLeafUnassigned(leaf)
18.            continue
19.        propagateConstraints(dGraph,
            leaf)
20.        removeVertex(dGraph, leaf)
```

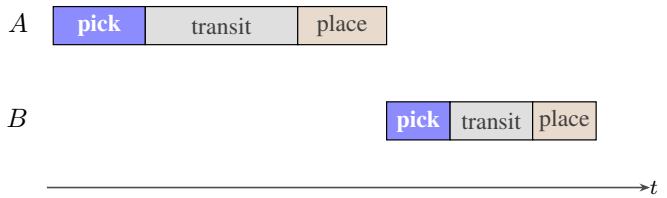
C. Task Queue

Once tasks are assigned to robots, we need to execute them. We assume that each task is associated with a robot r_i, an object o_j, a start time t_{start}, and a goal region $X_G \subseteq Q_i$, which is represented by the goal of object o_j (within some threshold).

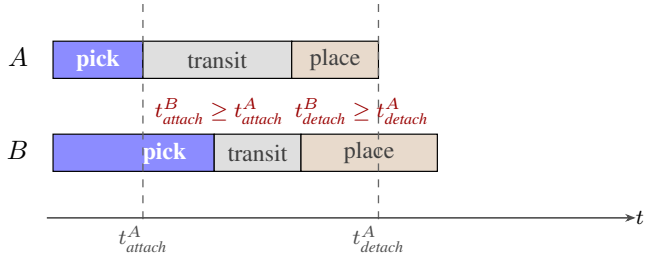
Centralized storage maintains the relationship between each robot and a queue of tasks for each robot. A robot is considered available if and only if its queue is empty. Tasks are executed in first-in-first-out order. The front task is invoked at each simulation step until it signals completion. The completion of the task triggers the removal, and the next task begins execution. If the queue is empty, the robot is assumed to be available again and ready for new task assignments.

D. Temporal Dependency Constraints

The high-level behavior in Alg. 1 removes a dependency vertex after the associated task has been finished. Afterwards,



(a) Without time constraints.



(b) With time constraints.

Fig. 4: Example timeline with pick/place time constraints for two robots A and B.

new leaf nodes are available for manipulation. This is a conservative approach, since a successor object can begin its planning before its predecessor has completed its full task sequence. Waiting for the full completion of the predecessor's task wastes time, during which a second robot could already begin approaching the next object.

To exploit this, each vertex in the dependency graph carries a temporal constraint that controls when temporal bounds are propagated to its successors. This is shown in Fig. 4. Without a temporal propagation, the successors become available only when the vertex is removed from the tree (Fig. 4a). With a temporal constraint, however, the attach and detach times are propagated (Fig. 4b). When the attach time t_{attach} is known, the successor can finish their pick phase as soon as the predecessor object is picked by another robot. Similarly, when the detach time t_{detach} is known, the successor can finish their place phase as soon as the predecessor object was placed. Note that we do not require those constraints to be present, in which case we either have to wait for the placement (if a dependency exists between objects) or no constraints are present (if no dependency exists). Having those lower bounds on the starting times of the respective phases is advantageous to obtain a tight scheduling without robots being idle.

IV. Generalized Disassembly Task

While CoMuDi can deal with arbitrary input tasks, we focus in this work on disassembly and define a generalized disassembly task. This task contains a sequence of subtasks as follows:

1) **Move to Object Grasp** Plan a path to a grasp configuration on the object surface with unconstrained orientation
2) **Attach**: Establish a kinematic coupling between end-effector and object.
3) **[Optional] Object Removal**: Advance through a set of object poses $T_1, \ldots, T_m$ using iterative IK with time scaling.

Algorithm 2: Generalized Disassembly Task

Input: robot, object, T_{goal}, t_{pick}, t_{place}
Output: success

1. $n_s \leftarrow$ |robot.states| `// Rollback checkpoint`
2. $n_a \leftarrow$ |object.attachData|
3. **if *not*** `moveToObject(robot, object,` t_{pick}`)` **then**
4. `rollback(`n_s`,` n_a`)`
5. **return** failure
6. `attach(robot, object)`
7. **if** *object.removalPath* $\neq \varnothing$ **then**
8. **if *not*** `followPath(robot, object.removalPath)` **then**
9. `rollback(`n_s`,` n_a`)`
10. **return** failure
11. **if** *object.insertionPath* $\neq \varnothing$ **then**
12. $T_{\text{goal}} \leftarrow T'_1 \cdot (T^{\text{obj}}_{\text{EE}})^{-1}$
13. **else**
14. $T_{\text{goal}} \leftarrow T_{\text{goal}} \cdot (T^{\text{obj}}_{\text{EE}})^{-1}$
15. **if *not*** `moveToGoal(robot,` T_{goal}`,` t_{place}`)` **then**
16. `rollback(`n_s`,` n_a`)`
17. **return** failure
18. **if** *object.insertionPath* $\neq \varnothing$ **then**
19. **if *not*** `followPath(robot, object.insertionPath)` **then**
20. `rollback(`n_s`,` n_a`)`
21. **return** failure
22. `detach(robot, object)`
23. **if *not*** `exit(robot)` **then**
24. `rollback(`n_s`,` n_a`)`
25. **return** failure
26. **return** success

4) **Move to Goal or Insertion Start** Plan a path to transport the object to its goal or insertion start pose T_{goal}.
5) **[Optional] Object Insertion**: Advance through a set of object poses $T'_1, \ldots, T'_k$ to guide the object into its goal position.
6) **Detach**: Terminate the kinematic coupling.
7) **Exit** Move the robot outside the active workspace to unblock any other robots in the scene.

Both the removal and insertion paths are precomputed from the assembly geometry and provided as input. Note that many algorithms exist to accomplish this [2, 34].

The pseudocode for the generalized disassembly task is described in Alg. 2. First, the current number of robot states (Line 1) and attachData (Line 2) are stored in case a task fails and the initial state has to be restored. This rollback and return failure behavior is repeated after every task that is run (Line 4–5, 9–10, 16–17, 20–21, 24–25).

The path to approach the object for the robot is calculated (Line 3). After the robot plans to the grasp position, the attachData is stored (Line 6). If the object has an associated

removal path (Line 7), the robot follows the path specified (Line 8). If the object has an insertion associated (Line 11), the desired end-effector transformation is calculated from the start of the path and the relative transformation between end-effector and object (Line 12) instead of the specified target (Line 14). The robot then moves to the calculated position (Line 15). If an insertion path is specified the robot follows the waypoints (Line 19). After the goal transformation is achieved, the robot detaches the object (Line 22) and leaves the area to make room for other robots (Line 23). The algorithm signals success after all subtasks are completed (Line 26).

A. Move Task

The move task plans a collision-free path from the robot's current configuration $q_{\text{start}} \in \mathcal{Q}$ to a specified goal pose $T_{\text{goal}} \in \mathcal{W}_{\text{free}}$ in the collision free workspace or a configuration $q_{\text{goal}} \in \mathcal{Q}$ in the robot's configuration space.

The pseudocode is given in Alg. 3. Given the robot's current initial configuration (Line 1), the planner retries the movement request for n_{retry} times (Line 2). If a pose is specified as the goal, a set of goal configurations is generated (Line 3), and the iteration is finished early if no goal states are feasible (Line 4–5). The planner is then queried to generate the motion (Line 6). If the planner is successful in finding a solution (Line 7), the result is committed to the robot's path storage (Line 8) and the task returns success (Line 10). If all attempts are exhausted, the task returns false (Line 11).

B. Attach and Detach Task

These tasks represent the grasping of an object by the robot. The attach task saves the homogeneous transformation between end-effector and object at time t_{attach}.

$$T_{\text{EE}}^{\text{obj}} = (T_{\text{world}}^{\text{EE}})^{-1} \cdot T_{\text{world}}^{\text{obj}} \tag{1}$$

Saving the homogeneous transform between end-effector and object at the time of execution allows for later reconstruction of the object position based on the robot pose, as described in Sec. V-A. This task does not move the robot to the object and solely tracks the relationship between robot and object at a specific time.

The tracking is done through the `attachData`, which resembles the association between robot and object at a specific time.

The detach task terminates the kinematic coupling by setting the time t_{detach} of the current attachData associated with the robot.

$$T_{\text{world}}^{\text{obj}}(t) = T_{\text{world}}^{\text{EE}}(t) \cdot T_{\text{EE}}^{\text{obj}} \quad \forall \quad t_{\text{attach}} \leq t \leq t_{\text{detach}} \tag{2}$$

Therefore, each attachData keeps track of:

- o_j: A reference to an object
- r_i: A reference to a robot
- t_{attach}: The attach time
- t_{detach}: the detach time
- $T_{\text{EE}}^{\text{obj}}$: A relative transformation between object and end-effector of the robot

C. Exit Task

After placing an object, the robot has to vacate the area to avoid obstructing subsequent operations by another robot. The exit task plans a path to a location in a specified safe zone in the workspace, where it does not obstruct any other robot. The configuration is set to the home position of the robot to guarantee a neutral starting position for the next task.

The goal used in the exit task is the final configuration until the robot is reassigned a new task. If the configuration intersects with a committed trajectory of another robot at any future time point, a collision occurs. Therefore, a potential exit configuration has to be verified against all future time points of other trajectories.

To check the candidate for validity, a linear sweep is performed from the start of the exit task until the largest committed time point of any robot.

The pseudocode to check a configuration for future collisions is described in Alg. 10. First, the lower time point of the time window is determined by the state passed to check for future collision (Line 1). Then, the upper time is determined by largest committed time point of any robot's trajectory (Line 2). While the time of the state is smaller or equal to the maximum time point (Line 3), the state time is increased incrementally by Δt (Line 4 and 5). Each increment, the state of the simulation is checked for collision (Line 6 and 7) and returns early on occurrence (Line 8). After the complete time window is passed, the state is validated (Line 9).

With the introduced algorithms, the complete exit tasks can be formulated in Alg. 5. The robot considers the current position as the start position for the exit task (Line 1). It tries up to a specified number of times (Line 2) to find an exit plan. It first starts with an empty set of possible goal states (Line 3). This set is filled with the specified number of valid exit configurations (Line 4). First a candidate is drawn from the specified exit region (Line 5). Since the robot might stay at the configuration until the simulation ends, the configuration has to be checked for future collisions (Line 6). After the candidate is verified, it is added to the possible goal states (Line 7). If no goal states could be generated a retry is triggered early (Line 8 and 9). The planner then tries to connect the start configuration to one of the possible exit configurations (Line 10). If it succeeds (Line 11), the exit path is committed to the planner's trajectory (Line 12) and returns successfully (Line 13). When all retries are exhausted, the exit task returns failure (Line 14).

D. Object Extraction Task

Objects that are geometrically interlocked with adjacent parts are hard to move freely through the workspace after grasping. Instead, they must follow a precomputed path consisting of a sequence of waypoint transforms $T_1, \ldots, T_m \in \text{SE}(3)$. These waypoints describe a collision-free trajectory through the surrounding geometry. This procedure is used both for extraction (removing an object from surroundings) and insertion (guiding an object into its goal position).

The follow path task advances through the waypoints using inverse kinematics. For each waypoint T_i, the required end-

Algorithm 3: Move Task

Input: robot, goal, t_{I}
Parameters: n_{retry}
Output: success

1. $\mathbf{q}_{\text{I}} \leftarrow$ `getConfiguration(robot)`
2. **for** $i \leftarrow 1$ **to** n_{retry} **do**
3. goals $\leftarrow$ `generateGoalStates(robot, goal)`
4. **if** *goals* $= \varnothing$ **then**
5. continue
6. path $\leftarrow$ `plan(robot,` $\mathbf{q}_{\text{I}}$`,` t_{I}`, goals)`
7. **if** *path* $\neq \varnothing$ **then**
8. `appendPath(robot, path)`
9. **return** true
10. **return** false

effector pose is computed from the waypoint and the current grasp transform:

$$T_i^{\text{EE}} = T_i \cdot (T_{\text{EE}}^{\text{obj}})^{-1} \tag{3}$$

The IK solver attempts to find a joint configuration $\mathbf{q}_i$ that realizes this end-effector pose.

The inverse kinematics problem for each waypoint can be stated as a constrained optimization. The full space-time waypoint problem is:

$$\begin{aligned} \min_{\mathbf{q}\in\mathcal{Q},\, t\in\mathbb{R}} \; \alpha t + (1-\alpha)\|q_i - q_{i-1}\|^2 \quad \text{s.t.} \quad & T_{\text{FK}}(\mathbf{q}) = T_i^{\text{EE}} \\ & \mathbf{q}_l \leq \mathbf{q} \leq \mathbf{q}_u \\ & t > t_{i-1} + \epsilon \\ & (\mathbf{q}, t) \in \mathcal{Q}_{\text{free}}(t) \end{aligned} \tag{4}$$

where $\mathcal{Q}_{\text{free}}(t)$ denotes the free configuration space at time t, determined by the committed trajectories of all other robots and objects.

Solving Eq. 4 jointly is intractable in practice, as the collision constraint couples the kinematic and temporal variables through the full simulation state. The problem is therefore decomposed into two sequential subproblems. First, the kinematic subproblem is solved independently

$$\min_{\mathbf{q}\in\mathcal{Q}} \; \|T_{\text{FK}}(\mathbf{q}) - T_i^{\text{EE}}\|^2 + \lambda\|q_i - q_{i-1}\|^2 \quad \text{s.t.} \quad \mathbf{q}_l \leq \mathbf{q} \leq \mathbf{q}_u \tag{5}$$

Second, the temporal subproblem finds the earliest collision-free time for the IK solution:

$$t_i = \min\{t \geq t_{i-1} + \epsilon \mid (\mathbf{q}_i, t) \in \mathcal{Q}_{\text{free}}(t),\; t - t_{i-1} < t_{\max}\} \tag{6}$$

by incrementally advancing in steps of Δt until a valid time is found or the maximum time increase $t_{\max}$ is exceeded. This decomposition sacrifices joint optimality for tractability. Configurations can exist that would be valid at an earlier time with a different $\mathbf{q}$ but are not discoverable by the sequential approach. In practice, the waypoint density of the precomputed paths is sufficient to ensure that the decoupled solution reliably finds valid states.

The pseudocode for object extraction is described in Alg. 4. The algorithm starts by getting the last committed trajectory

Algorithm 4: Object Extraction

Input: robot, object, path $= [T_1, \ldots, T_m]$
Parameters: $t_{\max}$, Δt
Output: success

1. $t \leftarrow$ `getLastStateTime(robot)`
2. **for** $T_i \in$ *path* **do**
3. $T_i^{\text{EE}} \leftarrow T_i \cdot (T_{\text{EE}}^{\text{obj}})^{-1}$
`// Solve IK, regrasping on failure`
4. **repeat**
5. $\mathbf{q}_i \leftarrow$ `solveIK(robot,` T_i^{EE}`)`
6. **if** $\mathbf{q}_i = \varnothing$ **then**
7. **if** ***not*** *object.allowRegrasp* **then**
8. **return** failure
9. `regrasp(robot, object,` t`)`
10. $T_i^{\text{EE}} \leftarrow T_i \cdot (T_{\text{EE}}^{\text{obj}})^{-1}$ `// Recompute with new grasp`
11. **until** $\mathbf{q}_i \neq \varnothing$
`// Find earliest valid time`
12. $t_{\text{start}} \leftarrow t$
13. **while** $t - t_{start} < t_{max}$ **do**
14. $s \leftarrow$ `createState(`$\mathbf{q}_i$`,` t`)`
15. **if** `isValid(s)` **then**
16. `robot.states.append(`s`)`
17. **break**
18. **else**
19. $t \leftarrow t + \Delta t$
20. **if** $t - t_{start} \geq t_{max}$ **then**
21. **if** ***not*** *object.allowRegrasp* **then**
22. **return** failure
23. `regrasp(robot, object,` t`)`
24. **redo** current waypoint T_i
25. **return** success

time point of the robot (Line 1). It then iterates through each waypoint provided (Line 2). First it calculates the end-effector transformation from the desired object path (Line 3). It then solves the inverse kinematics problem for that transformation (Line 5) to find the next configuration. If no valid configuration is found (Line 6), a regrasp can be attempted if allowed (Line 7–9). When the object does not permit regrasps, a failure is returned (Line 8). After the regrasp is successful, the desired end-effector transformation is updated with the new relative transformation between end-effector and object (Line 10). This procedure is repeated until a valid configuration is found (Line 11).

After a possible configuration is found, the last committed trajectory time point is stored (Line 12). This time point is then used to iterate over a time window, which is limited by an upper bound (Line 13). The solution from the inverse kinematics is set to the relevant time point (Line 14) and validated for collision (Line 15). If no collision occurs, the state is committed to the robot's trajectory (Line 16) and continues to the next inverse kinematics problem (Line 17). In case of collision, the time point is increased by a time delta Δt (Line 19), until either a valid time point is found or

Algorithm 5: Exit Task

Input: robot
Parameters: n_{retry}, numberOfExitConfigurations
Output: success

1. $\mathbf{q}_{\text{start}} \leftarrow$ `getPose(robot)`
2. **for** $i \leftarrow 1$ **to** n_{retry} **do**
3. goalStates $\leftarrow \varnothing$
4. **for** $j \leftarrow 1$ **to** *numberOfExitConfigurations* **do**
5. $\mathbf{q}_{\text{exit}} \leftarrow$ `sampleExitRegion()`
6. **if** `checkFutureCollision(`q_{exit}`)` **then**
7. `append(goalStates,` $\mathbf{q}_{\text{exit}}$`)`
8. **if** *goalStates* $= \varnothing$ **then**
9. **continue**
10. path $\leftarrow$ `plan(robot,` $\mathbf{q}_{\text{start}}$`, goalStates)`
11. **if** *path* $\neq \varnothing$ **then**
12. `appendPath(robot, path)`
13. **return** true
14. **return** false

the time window is exhausted.

After the time window is over (Line 20) and no solution was found, a regrasp is performed (Line 23) and the current waypoint is retried again (Line 24). If no regrasp is permitted, the planner instead fails (Line 22) after no valid solution was found.

When all waypoints find a suitable configuration and timepoint, the algorithm returns success (Line 25).

V. Collision Checking and Grasp Generation

Next we need to be able to generate grasp positions and do collision checking with existing robots and objects.

A. Collision Checking

All tasks rely on a collision checking algorithm, which checks if a certain configuration is in collision at a specific time point. The pseudocode for collision checking is described in Alg. 6. First, the associated time with the state is acquired (Line 1). Then, the simulation state is set with Alg. 8 (Line 2). Then the new configuration is set (Line 3). Since it is unknown if an object is attached at that time point to the robot, each object (Line 4) is checked if it is attached to the desired robot. First, the relevant attachment data for that time point is searched with Alg. 7 (Line 5). If the robot associated with the attachData matches the selected robot (Line 6), the current end-effector pose is saved (Line 7) and combined with the saved relative transform (Line 8) between object and robot from the attachData to the actual pose of the object at that time point (Line 9). Then the pose of the object is set (Line 10) and the loop exits early (Line 11). After each object and robot is set, the algorithm returns if a collision is detected by the collision detector (Line 12).

Algorithm 6: Collision Check

Input: q_{new}, $robot_{\text{selected}}$
Output: boolean

1. $t_{\text{at}} \leftarrow$ `getTime(`q_{new}`)`
2. `setSimulationState(`t_{at}`)`
3. `setConfiguration(`$\text{robot}_{\text{selected}}$`,` q_{new}`)`
4. **for** $o_j \in \mathcal{O}$ **do**
5. data $\leftarrow$ `findAttachmentData(`o_j`,` t_{at}`)`
6. **if** o_j*.robot* $==$ $robot_{selected}$ **then**
7. eeTF $\leftarrow$ robot.eeTF
8. eeToObjTF $\leftarrow$ data.eeToObjTF
9. $objTF \leftarrow eeTF * eeObjTF$
10. `setPose(obj, objTF)`
11. **break**
12. `return isSimInCollision()`

Algorithm 7: Find Attachment Data

Input: object, t_{at}
Output: Attachment Data

1. attachData $\leftarrow$ `getAssociatedAttachData(object)`
2. **assert** attachData $\neq \varnothing$
3. firstAttachTime $\leftarrow$ `attachData.front().attachTime`
4. **if** $t_{at} <$ *firstAttachTime* **then**
5. `return attachData.front()`
6. lastDetachTime $\leftarrow$ `attachData.back().detachTime`
7. **if** $t_{at} >$ *lastDetachTime* **then**
8. `return attachData.back()`
9. **foreach** $data \in attachData$ **do**
10. **if** $t_{at} > t_{attach} \wedge t_{at} < t_{detach}$ **then**
11. `return` data

1) Find Attachment Data: Since objects can be attached to different robots at different time points, information about the attachment time, detachment time and relative transform between end-effector and object has to be saved to restore the complete state at any time point. Attachment data is stored in chronological order of occurrence.

The pseudocode to find the attachData at a specific time point is described in Alg. 7. First, the associated data storage is selected (Line 1). It then verifies that the object was actually moved (Line 2). Since the data is stored in chronological order, the first attach time is used as an early return to indicate that the object is still in its initial pose (Line 3–5). The same behavior is true for every time point after the last detach time point. The last detach time (Line 6) is compared against the timepoint (Line 7) to determine if an early return (Line 8) is feasible. Both of the early returns return their attachData as a comparison against the timepoint to allow to check if the object should be placed in the start or goal pose.

If neither condition is met, a linear sweep across the data (Line 9) is combined with a range check on the time point with

Algorithm 8: Set Simulation State

Input: t_{at}
1. **for** *obj* $\in$ *objects* **do**
2. data $\leftarrow$ `findAttachmentData(obj, t_at)`
3. robot $\leftarrow$ `getRobot(data)`
4. time $\leftarrow$ `determineTimePoint(data, t_at)`
5. q_{lower}, $q_{\text{upper}} \leftarrow$ `findBoundingStates(robot, time)`
6. $q_{\text{inter}} \leftarrow$ `interpolate(q_lower, q_upper, time)`
7. `setConfiguration(robot, q_inter)`
8. eeTF $\leftarrow$ robot.eeTF
9. eeToObjTF $\leftarrow$ data.eeToObjTF
10. $objTF \leftarrow eeTF * eeObjTF$
11. `setPose(obj, objTF)`
12. **for** *robot* $\in$ *robots* **do**
13. q_1, $q_2 \leftarrow$ `findBoundingStates(robot, t_at)`
14. $q_{\text{inter}} \leftarrow$ `interpolate(q_1, q_2, time)`
15. `setConfiguration(robot, q_inter)`

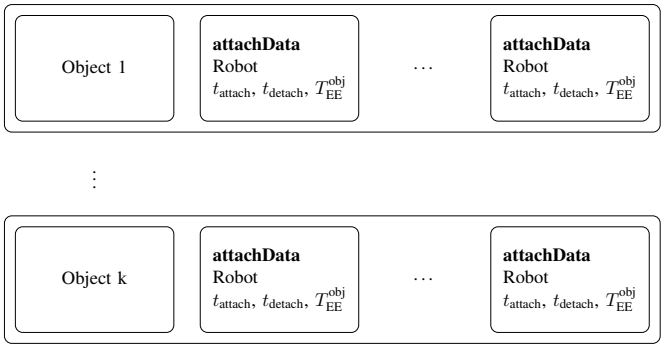


Fig. 5: Storage representation of attachment data

a time window, consisting of the attach and detach time (Line 10). The relevant attachData is then finally returned (Line 11).

2) Setting the Simulation State: To execute collision checking at a specific time point, every object and robot has to be moved to the correct configuration. We call this the setting of the simulation state. The pseudocode to set the simulation state is described in Alg. 8. First, the algorithm iterates over each object to set their pose (Line 1). The relevant attachment data for each object is located (Line 2), as outlined in Alg. 7. The time step is categorized as either prior to, within or subsequent to the attachment data (Line 4). If the time step is prior to the first attachment data, the object has not yet been moved and remains at the starting position. If the time step falls within the time period of an attachment, the time itself is utilized to set the pose of the robot. Conversely, if the time step is larger than the last time period, the detach time of the last attachment data is used for the pose. The robot's states which lie right before and after the time point are identified (Line 5). Then, the state is interpolated (Line 6) and the robot's configuration is set (Line 7) for the time point. This is done, so the end-effector transformation can be acquired (Line 8). This is used with the relative end-effector-object transformation (Line 9) to reconstruct the object's world transformation (Line 10). The object is then set to that pose (Line 11).

After each object is placed to the correct transformation, each robot can be set (Line 12). Now the input time is used to determine the bounding states (Line 13) and interpolation (Line 14) like before. Then the robot is set to the interpolated configuration (Line 15).

Algorithm 9: Grasp Pose Generation

Input: robot, object
Parameters: maxTryNumber, maxNumGoals
Output: goalStates
1. goalStates $\leftarrow \varnothing$
2. **for** $i \leftarrow 0$ **to** *maxTryNumber* **do**
3. **if** $|goalStates| \geq maxNumGoals$ **then**
4. **return** goalStates
5. surfacePoint $\leftarrow$ `sampleSurface(object)`
6. q $\leftarrow$ `solveIK(robot, surfacePoint)`
7. **if** $q = \varnothing$ **then**
8. continue
9. **if** *not* `checkFutureCollision(q)` **then**
10. continue
11. `append(goalStates, q)`
12. **return** goalStates

B. Grasp Pose Generation

Each surface sample provides a target position for the inverse kinematic problem of the end-effector. Since the omnidirectional gripper can approach from any orientation, the orientation is not constrained during solving and a random orientation is sampled for each IK query. The inverse kinematics solver uses different restarts with random initial guesses sampled uniformly within the robot's configuration space $\mathcal{Q}$.

The goal state generation pseudocode is described in Alg. 9. First, an empty goalStates set is initialized (Line 1). Then, the algorithm tries up to a certain number of tries (Line 2) to generate at most a specified number of possible goal states (Line 3) and return early if the amount is met (Line 4). A point on the object is sampled (Line 5) and used as the target position for the inverse kinematics problem (Line 6). If no valid configuration can be found (Line 7), the next iteration begins (Line 8). When a valid configuration is found, it is only kinematically feasible and has to be checked to be free in the future (Line 9). After the state was validated it is added to the set of possible goal states for the planner (Line 11). The algorithm returns the set of all found goal state candidates (Line 12).

1) Check Future Collisions: Not every IK solution represents a valid grasp configuration. A candidate is only accepted if it passes a future collision check with Alg. 10. This prevents selecting grasp poses that are feasible but invalidated by other robots' trajectories at different time points.

2) Surface Sampling: A grasp can occur on any point on the surface of the object since the end-effector is modeled as an omnidirectional gripper, as described in the first assumption

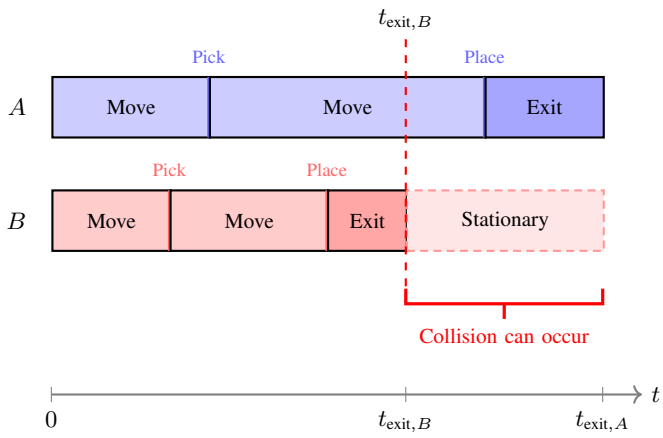


Fig. 6: Timeline illustration for future collision check

Algorithm 10: Check Future Collision

Input: robot, $\mathbf{q}$, t_{at}
Parameters: Δt
Output: validConfiguration

```
t ← t_at
t_max ← getMaxTrajectoryTime()
while t ≤ t_max do
    t ← t + Δt
    setSimTime(t)
    setConfiguration(robot, q)
    if isInCollision() then
        return false
return true
```

in Sec. III-A. The grasp pose generation produces a set of possible candidates in $\mathcal{Q}$ for the motion planner. These candidates are generated by sampling contact points on the surface of the object and then used as the target for inverse kinematics.

A point on the object surface serves as the target position for the end-effector.

The object surface is considered to be described as a mesh. Each mesh consists of vertices, edges and faces. Vertices describe positions in the space. Edges are connections between vertices. A closed set of edges is called a face. In this case, the faces are considered to be triangle faces, resulting in a triangle mesh.

For mesh objects, a face is selected with probability proportional to its area to ensure a uniform distribution of sample points across the surface. The cumulative area distribution over all faces is computed and a face is selected via inverse transform sampling.

$$A_i = \frac{1}{2}\|(v_1 - v_0) \times (v_2 - v_0)\| \tag{7}$$

The probability of selecting a face i from a mesh with N faces

$$P(i) = \frac{A_i}{\sum_{j=1}^{N} A_j} \tag{8}$$

Then a uniform sample from the distribution is used to select a face.

Within the selected face, a point is sampled using barycentric coordinates with two uniform random variables $r_1, r_2 \in [0, 1]$:

$$\mathbf{p} = (1 - r_1)\mathbf{v}_0 + r_1(1 - r_2)\mathbf{v}_1 + r_1 r_2 \mathbf{v}_2 \tag{9}$$

where $\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2$ are the faces' vertices. The result is transformed to world coordinates using the object's current pose.

VI. Evaluation

To evaluate our disassembly planner, we perform an evaluation on six scenarios shown in Fig. 7. For each scenario, we vary the number of robots from 1 to 9 to showcase the robustness of the planner to different team sizes and to demonstrate the optimal number of robots required for a specific scenario. Finally, we compare ST-RRT* as individual planner to two alternatives, namely RRT* with 5s and 10s time bounds, respectively.

A. Implementation Details

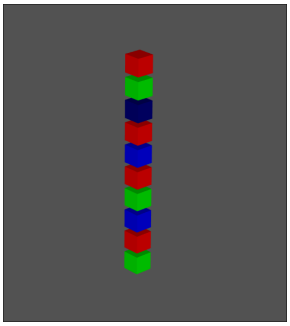
CoMuDi is implemented using the Open Motion Planning Library (OMPL) [33]. Both RRT* [16] and ST-RRT* [10] are used with an optimization objective to minimize the arrival time. The arrival time is the last time point of a trajectory for one planner query and is not the same as the makespan, which is the total time needed to complete the disassembly process, introduced in the problem statement in Sec. III-A. A maximum computation time $t_{\text{timelimit}}$ of 10s is used as a termination condition for the planner. After a solution is found shortcutting and B-spline smoothing are applied to the path.

B. Parameters

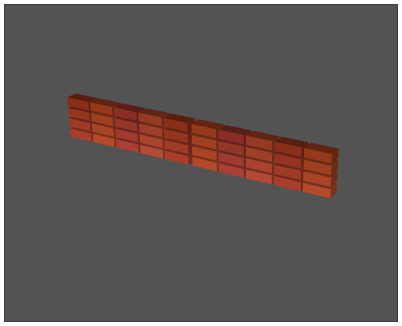
The tunable parameters of CoMuDi are set to the following values. The maximum number of surface-sampling and inverse-kinematics attempts during grasp pose generation (Alg. 9) is set to **maxTryNumber**= 1000. The maximum number of goal configurations collected before grasp pose generation returns (Alg. 9) is set to **maxNumGoals**= 5. The time increment for forward collision sweeps in future-collision checking (Alg. 10) is set to $\Delta t = 0.1$. For planning, we use $n_{\text{retry}} = 3$ for maximum number of planning attempts for the move and exit task. Further, for the exit task, we set **numberOfExitConfigurations** to 10. For the object extraction task, we set $t_{\max} = 10$. For the IK solver, we use $\epsilon = 0.1$ (Minimum time separation required between consecutive waypoint states) and $\lambda = 1$ (Weight on joint-displacement regularization).

C. Benchmark Hardware

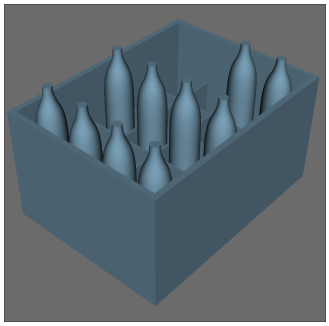
All runs are performed on a Dell XPS 15 9530 laptop, equipped with an Intel Core i7-13700H processor. Each trial is pinned on performance core 0 of the CPU with a stable clock speed between 4.7 and 4.8 GHz under sustained load. Runs were executed sequentially since the CPU frequency varied for each core under load if multiple instances ran in parallel. This ensures comparable results within the benchmarks.



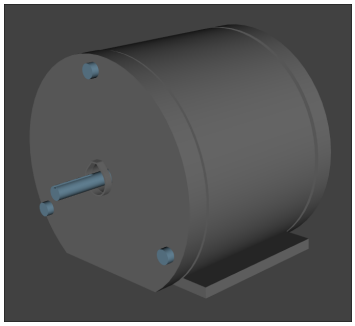
(a) Tower (10 Components)



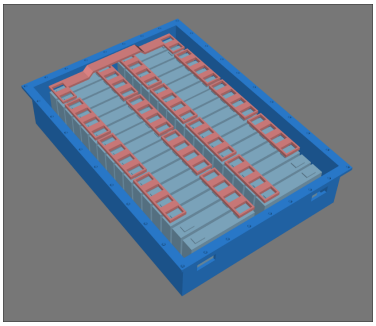
(b) Wall (40 Components)



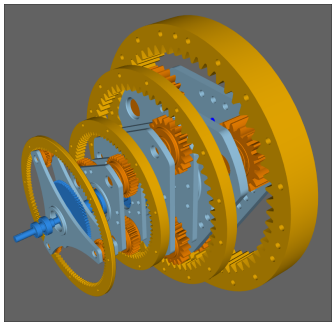
(c) Crate (10 Components)



(d) Motor (13 Components)



(e) Battery (49 Components)



(f) Gearbox (33 Components)

Fig. 7: Scenarios used in the disassembly experiments.

D. Multi-Robot Disassembly Benchmark

For each scenario, we report on computation time, makespan (total time elapsed from start to finish), Pareto front (makespan versus computation time), scheduling timeline, and analyze the failure rate.

1) Computation Time and Makespan: The first metric we evaluate is computation time and makespan. The computation time captures the cumulative time consumed across all queries needed to create the sequence, while the makespan captures the total time required to complete the planned disassembly sequence. The results are shown in Fig. 8. For both plots, we report the median with the shaded bands denoting the interquartile range[1] across runs. The makespan is expected to decrease, while the execution time per task is expected to grow as individual assignments take place in an increasingly populated workspace. It can be observed that this remains true for all six scenarios, with makespan decreasing and computation time increasing.

2) Pareto Front: While minimizing makespan is a high priority for disassembly tasks, it comes at the cost of increasing computation time. To better understand this trade-off, we visualize those two metrics in a Pareto front plot in Fig. 9. The plot shows one sample per robot count, with the number of robots written inside. A blue line connects all samples which are lying on the pareto frontier, meaning that no other sample is dominating it. A sample dominates another sample, if both makespan and computation time are lower. It can be observed that all samples are on or are lying close to the pareto frontier, meaning that varying the number of robots clearly trades-off makespan and computation time. We can also observe that a number of 3 to 5 robots is often a good compromise between those two metrics.

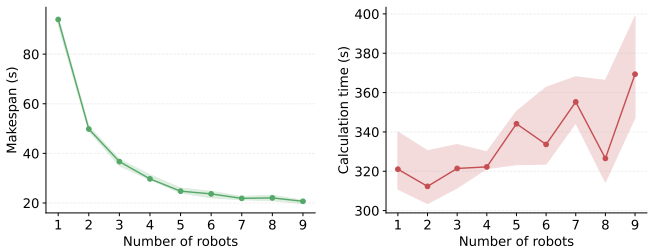
3) Scheduling Timeline: Minimizing makespan is often influenced by the idle time of the robots, with less idle time often being preferable. We visualize this metric for all 9 robots in Fig. 10. For each robot, we visualize how much time is spent being idle (white), executing a pick move (blue), a place move (orange), a pull move (red), or an exit move (brown). It can be seen that idle time is minimized except when the nature of the scenario prevents this. For example the crate scenario is highly parallel, in that robots have no tasks left once they solve their assigned task.

4) Failure Rate Analysis: Finally, we conduct a failure rate analysis as shown in Fig. 11. This analysis gives a more detailed insight into our planner, and shows how failures are changing with the number of robots. In detail, we plot the percentage of success (green) for each robot count, together with the percentages of four failure modes: exit mode failure (light green), pull mode failure (yellow), plan to object failure (blue) and plan to goal failure (purple). Those failure modes do not stop the algorithm, but they represent wasted computation time, which we want to minimize for better computation time.

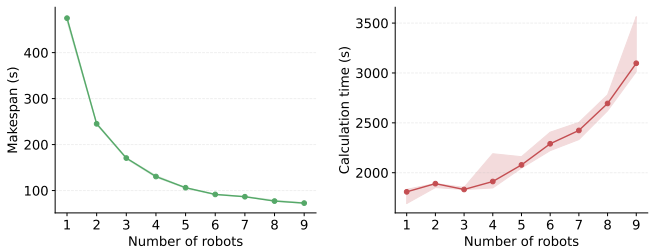
E. Individual Planner Comparison

This section compares RRT* and ST-RRT* in the context of the tower scenario. We compare makespan and success rate

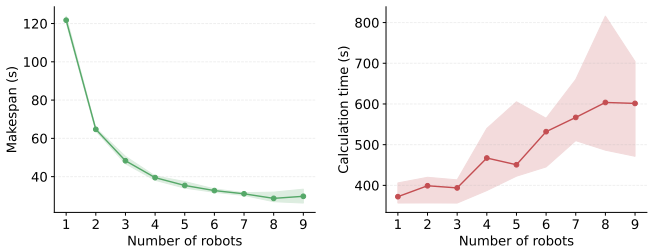
[1]The interquartile range marks the area in which 50% of the samples fall.



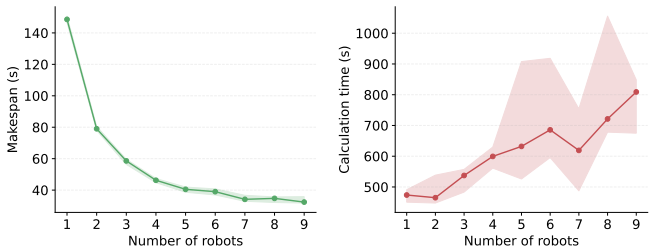
(a) Makespan and Computation Time for Tower



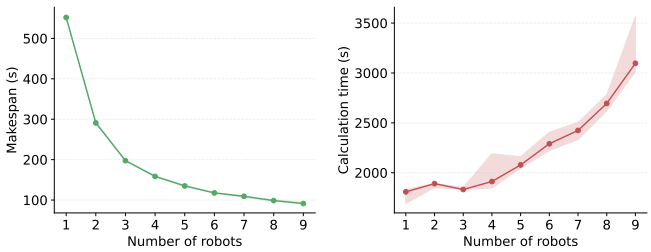
(b) Makespan and Computation Time for Wall



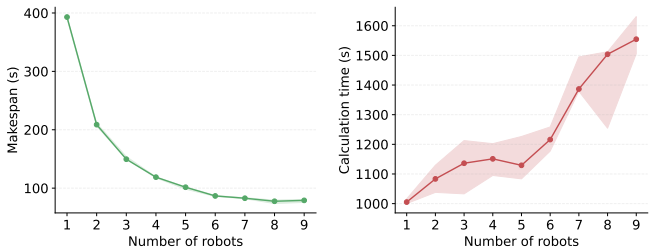
(c) Makespan and Computation Time for Crate



(d) Makespan and Computation Time for Motor



(e) Makespan and Computation Time for Battery



(f) Makespan and Computation Time for Gearbox

Fig. 8: Scaling behavior for the two metrics of makespan and computation time for different robot counts in each disassembly scenario.

while varying the number of robots from one to nine. Each run has a timeout of 1000 seconds, 10 runs per robot count, and an individual planner timeout of 10 seconds.

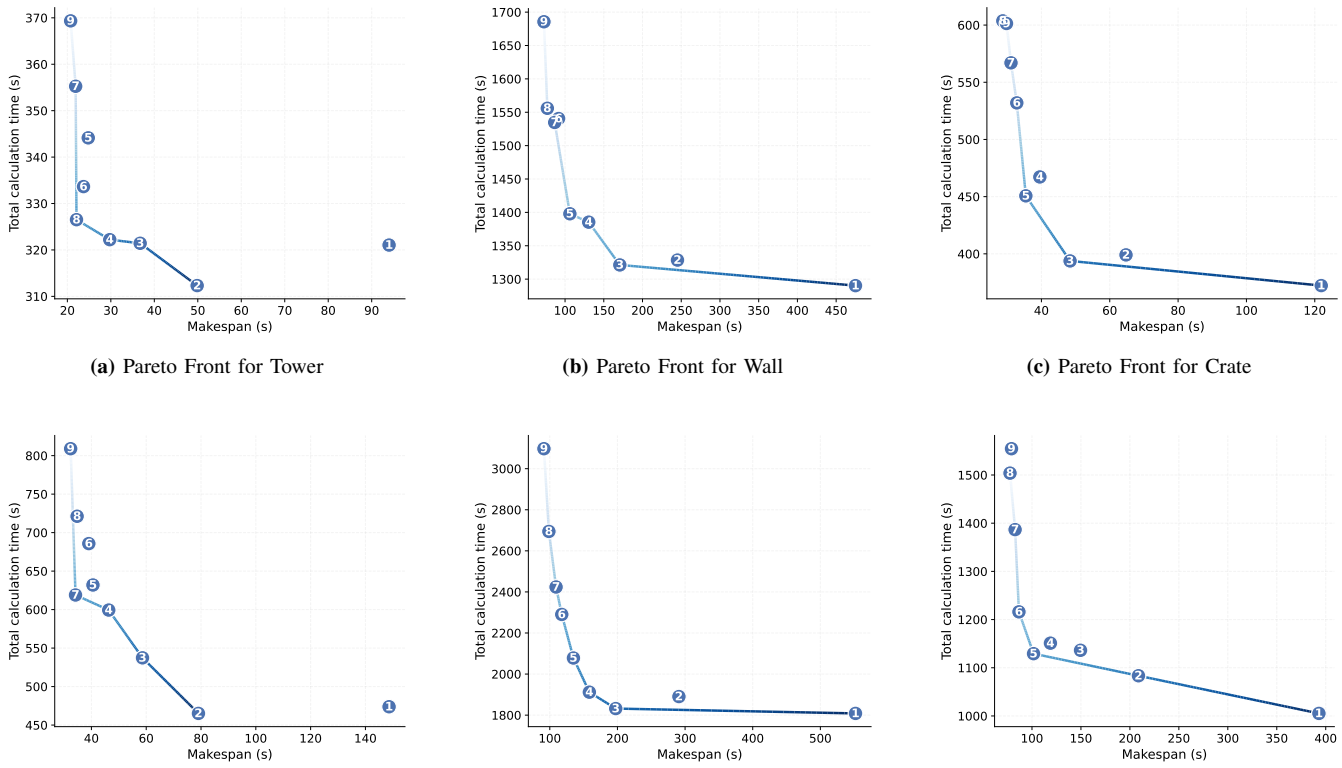
Since RRT* can only operate within a fixed time space [10], we evaluate two time windows. A 5 second window is used as a tight bound with little room for detours, but easier to optimize due to the smaller state space. A 10 second window allows for greater flexibility in space-time, enabling detours to be taken to avoid robots. Both time windows were empirically selected based on the convergence behavior of ST-RRT* in Fig. 8. These bounds are otherwise not known a priori.

The scaling of the makespan for all three methods is shown in Fig. 12. With a 5 second time window, the makespan for RRT* is the highest at the start, with 168s for one robot, 82s for three robots, until it closes the gap with 26s for five robots. After five robots, the makespan matches or outperforms that of the ten-second time window. With the 10s time window, the performance is similar to ST-RRT* at the beginning, from 99 seconds for one robot to 27 seconds for five robots. Afterwards, the makespan increases again, fluctuating between 31 and 36 seconds for six to nine robots. ST-RRT* consistently outperforms the makespan of the other two methods. It starts at 94 seconds with one robot and falls monotonically to 20 seconds with nine robots.

The success rate of the three methods is shown in Fig. 13. RRT* with a 10s window starts at 100% for one robot, but drops rapidly to around 30% for three or more robots. The 5s window for RRT* has a success rate of around 50%, dropping to 20% with four and six robots. ST-RRT* achieves a success rate of 100% throughout.

VII. Discussion

The experiments show two important results. First, CoMuDi can successfully coordinate multi-robot teams to disassemble large-scale objects with up to 49 pieces. Overall, we were able to show that CoMuDi can lower the makespan with an increase in the number of robots. This is a clear indicator that CoMuDi can efficiently orchestrate robot teams even when the number of robots increases, thereby decreasing the movement space for individual robots. While the computation time increases to coordinate larger robot teams, we have been able to show that almost all computations are close to the Pareto front, meaning we use the available computation time efficiently. Additionally, our results show that planning is a trade-off between makespan



(a) Pareto Front for Tower **(b)** Pareto Front for Wall **(c)** Pareto Front for Crate

(d) Pareto Front for Motor **(e)** Pareto Front for Battery **(f)** Pareto Front for Gearbox

Fig. 9: Pareto front for each disassembly scenario.

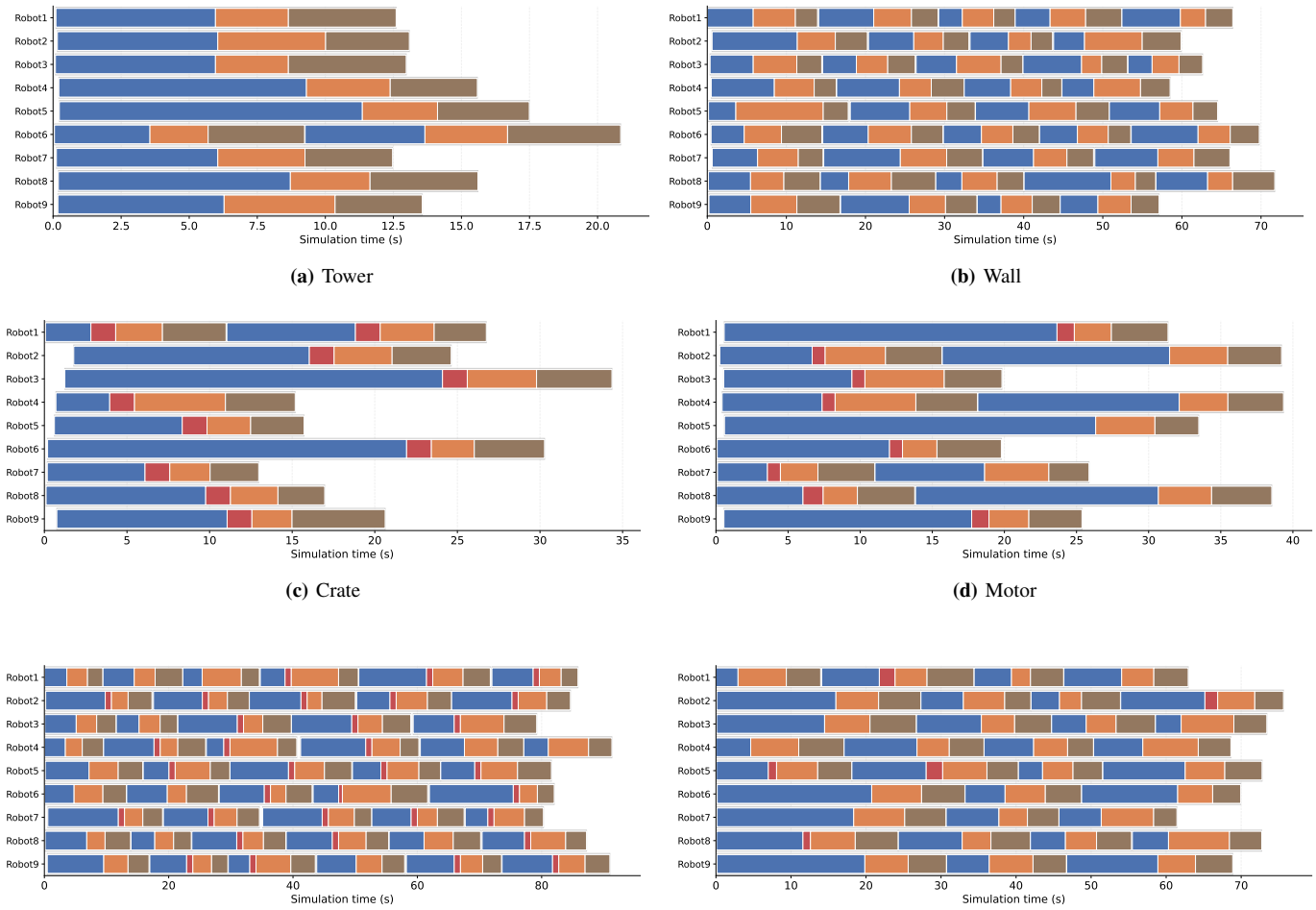
and computation time, thereby allowing users of the system to carefully weigh the number of robots needed to accomplish a certain disassembly task.

The second result concerns the comparison between ST-RRT* and RRT*. This comparison shows the importance of designing a planner to operate directly in space-time, rather than adapting an existing planner to incorporate this feature. While RRT* with a ten second window achieves a comparable makespan to ST-RRT* for one to five robots, RRT* struggles to find competitive solutions beyond this. The wider time window provides flexibility for detours, but the resulting state space becomes too large to explore and optimize once the workspace becomes crowded with other robots. The decline in success rate with an increasing number of robots reflects the same limitation. ST-RRT* avoids this issue by allowing the time bound to shrink or grow on demand. It combines the makespan quality of the wider window at low robot counts with the sampling density of the narrower window at high robot counts, outperforming both across the full range. The adaptive bound comes at a modest computational cost, since additional work is required to determine the bounds at runtime. This result shows that CoMuDi with ST-RRT* is a particularly efficient combination and leads consistently to successful results with a low makespan.

A. Limitations

While the results show that CoMuDi is approaching near optimal results in coordinating multiple robots, there are still some improvements possible to our framework. In particular:

- **Depth-Aware Assignment Policy** : Currently, CoMuDi assigns the next available object to a robot without considering the nodes currently executing in the dependency graph. Taking this into account would balance the assembly more evenly throughout the workspace. In the motor and gearbox scenario, a better assignment policy could enable disassembly from two sides rather than relying on pure chance. This would prevent dependency chains with long pick phases, which slow down the makespan. To tackle this, we need to consider the depth in the dependency graph, the movement of other robots around the object, and take the available workspace into account.
- **Clearance Cost** : The object extraction task considers each pull as a constrained motion scaled in time. This pull phase does not consider the proximity of obstacles or the configurations of other robots planned earlier or later. A more sophisticated approach would include a clearance cost to avoid critical areas so that the robot would proactively avoid obstructions for future tasks while solving for the current pull.
- **Grouping of Objects** : Another improvement concerns objects that are in close proximity. For example, the screws of the motor or the gears of the gearbox. Such



(a) Tower **(b)** Wall

(c) Crate **(d)** Motor

(e) Battery **(f)** Gearbox

Fig. 10: Scheduling timelines of the disassembly process for each scenario. Tasks are color coded as Pick, Place, Pull, and Exit.

objects could be grouped so that the IK solver can try to fit as many manipulators as possible into the workspace simultaneously.

- **Global Optimality** : Another bottleneck is the division into separate planning problems. Since there is no overarching planning objective, this can result in an unfavorable starting position in subsequent planning problems. The planner should be able to detect this and sacrifice local optimality for global optimality.

VIII. Conclusion

We presented the coordinated multi-robot disassembly (CoMuDi) task and motion planning framework for disassembly tasks. CoMuDi integrates scale-invariant sampling [2] for object extraction together with time-optimal motion planning [10] into a coherent framework for multi-robot coordination [12]. To ensure CoMuDi can robustly and efficiently solve disassembly tasks, we define generalized disassembly tasks, propagate temporal constraints, and develop a temporal collision checking framework.

In six disassembly scenarios, we were able to show that CoMuDi can reliably solve disassembly tasks, even when those tasks require long-horizon planning with up to 9 robots and up to 49 objects in a single assembly. The experiments show that CoMuDi can successfully optimize makespan, especially when the number of robots is increased. The scheduling timelines show that robots minimize their idle time, and the ST-RRT* comparisons with RRT* show that CoMuDi with ST-RRT* achieves the best success rate and lowest makespan.

While some limitations remain, the proposed planner already shows near optimal behavior and can be used to coordinate the motion of multi-robot teams operating in tight proximity to each other. This makes CoMuDi an essential part for any general-purpose disassembly system to tackle future applications in recycling and remanufacturing.

References

[1] A. Andreychuk, K. Yakovlev, P. Surynek, D. Atzmon, and R. Stern, "Multi-agent pathfinding with continuous time," *Artificial Intelligence*, vol. 305, p. 103662, 2022.

[2] S. B. Bayraktar, A. Orthey, and M. Toussaint, "Scale-invariant sampling in multi-arm bandit motion planning for object extraction," in *World Symposium on the Algorithmic Foundations of Robotics*, 2026.

[3] M. Bennewitz, W. Burgard, and S. Thrun, "Finding and optimizing solvable priority schemes for decoupled path planning techniques for teams of mobile robots," *Robotics and Autonomous Systems*, vol. 41, no. 2-3, pp. 89–99, 2002.

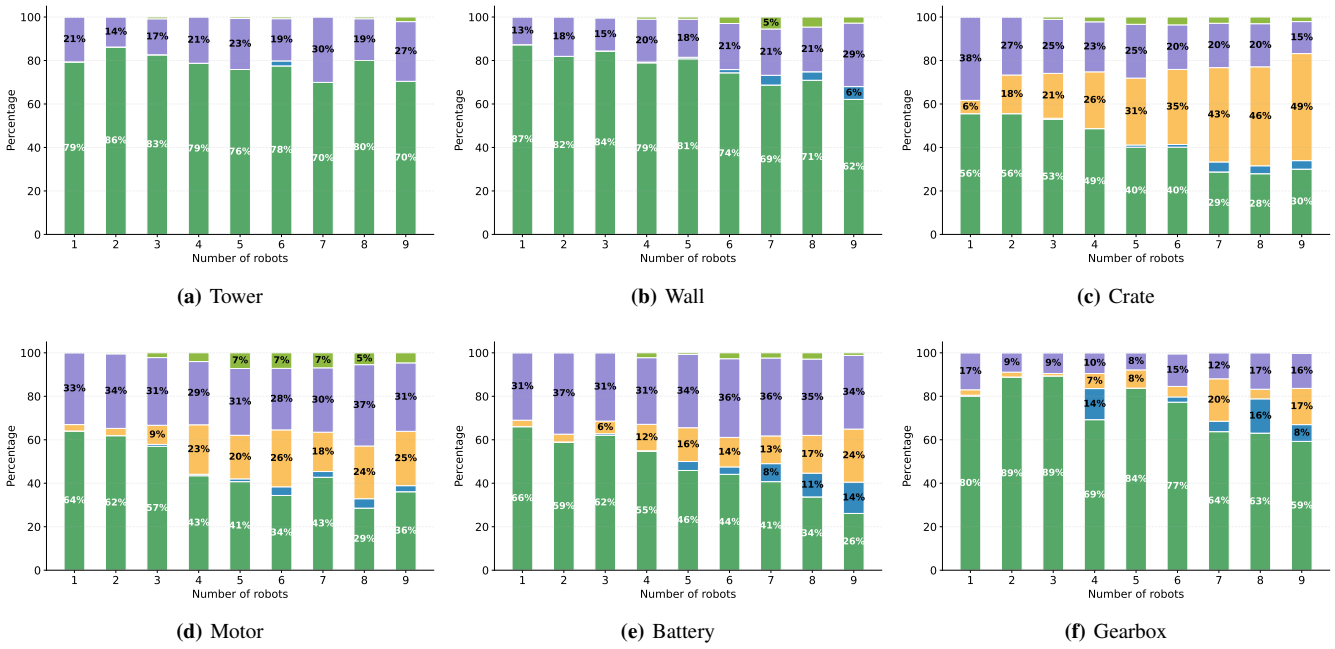


Fig. 11: Failure rates across the different disassembly scenarios. Success Exit fail Pull fail Plan to object fail Plan to goal fail

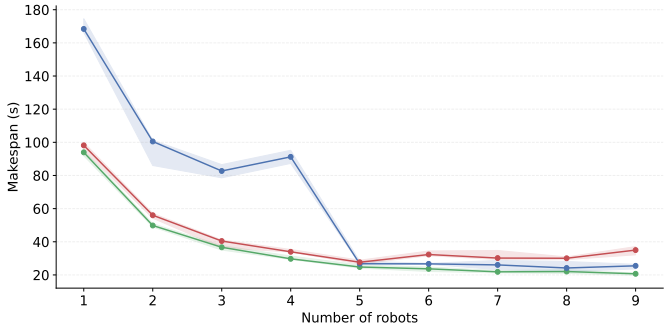


Fig. 12: Makespan comparison between ST-RRT* and RRT*, showing ST-RRT* , RRT* 5s, and RRT* 10s.

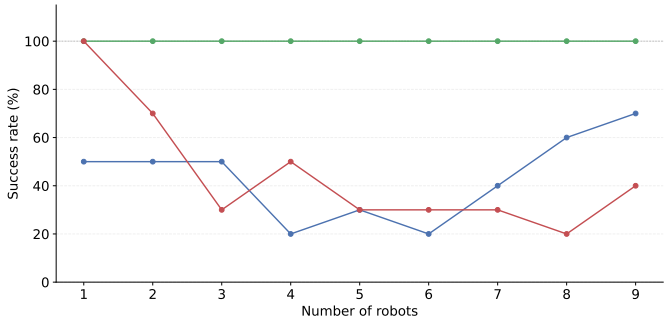


Fig. 13: Success rate comparison of ST-RRT* and RRT*, showing ST-RRT*, RRT* 5s, and RRT* 10s.

[4] M. Čáp, P. Novák, A. Kleiner, and M. Selecký, "Prioritized planning algorithms for trajectory coordination of multiple mobile robots," *IEEE Transactions on Automation Science and Engineering*, vol. 12, no. 3, pp. 835–849, 2015.

[5] J. Chen, J. Li, Y. Huang, C. Garrett, D. Sun, C. Fan, A. Hofmann, C. Mueller, S. Koenig, and B. C. Williams, "Cooperative task and motion planning for multi-arm assembly systems," *arXiv preprint arXiv:2203.02475*, 2022.

[6] M. Erdmann and T. Lozano-Pérez, "On multiple moving objects," *Algorithmica*, vol. 2, no. 1, pp. 477–521, 1987.

[7] C. R. Garrett, T. Lozano-Pérez, and L. P. Kaelbling, "Pddlstream: Integrating symbolic planners and blackbox samplers via optimistic adaptive planning," in *Proceedings of the international conference on automated planning and scheduling*, vol. 30, 2020, pp. 440–448.

[8] C. R. Garrett, R. Chitnis, R. Holladay, B. Kim, T. Silver, L. P. Kaelbling, and T. Lozano-Pérez, "Integrated task and motion planning," *Annual Review of Control, Robotics, and Autonomous Systems*, vol. 4, pp. 265–293, 2021.

[9] M. Ghallab, C. Knoblock, D. Wilkins, A. Barrett, D. Christianson, M. Friedman, C. Kwok, K. Golden, S. Penberthy, D. Smith, Y. Sun, and D. Weld, "Pddl - the planning domain definition language," Yale Center for Computational Vision and Control, Tech. Rep. CVC TR-98-003/DCS TR-1165, 08 1998.

[10] F. Grothe, V. N. Hartmann, A. Orthey, and M. Toussaint, "ST-RRT*: Asymptotically-optimal bidirectional motion planning through space-time," in *2022 International Conference on Robotics and Automation (ICRA)*, 2022, pp. 3314–3320.

[11] V. N. Hartmann and M. Toussaint, "Towards computing low-makespan solutions for multi-arm multi-task planning problems," in *ICAPS Workshop on Planning and Robotics (PlanRob)*, 2023.

[12] V. N. Hartmann, A. Orthey, D. Driess, O. S. Oguz, and M. Toussaint, "Long-horizon multi-robot rearrangement planning for construction assembly," *IEEE Transactions on Robotics*, vol. 39, no. 1, pp. 239–252, 2023.

[13] K. Hauser and J.-C. Latombe, "Multi-modal motion planning in non-expansive spaces," *The International Journal of Robotics Research*, vol. 29, no. 7, pp. 897–915, 2010.

[14] D. Hsu, R. Kindel, J.-C. Latombe, and S. Rock, "Randomized kinodynamic motion planning with moving obstacles," *The International Journal of Robotics Research*, vol. 21, no. 3, pp. 233–255, 2002.

[15] L. P. Kaelbling and T. Lozano-Perez, "Hierarchical task and motion planning in the now," in *2011 IEEE International Conference on Robotics and Automation*, 2011, pp. 1470–1477.

[16] S. Karaman and E. Frazzoli, "Sampling-based algorithms for optimal motion planning," *The International Journal of Robotics Research*, vol. 30, no. 7, pp. 846–894, 2011.

[17] M. Kheder, M. Trigui, and N. Aifaoui, "Disassembly sequence planning based on a genetic algorithm," *Proceedings of the Institution of Mechanical Engineers, Part C: Journal of Mechanical Engineering Science*, vol.

229, 11 2014.
[18] E. Kongar and S. M. Gupta, "Disassembly sequencing using genetic algorithm," *The International Journal of Advanced Manufacturing Technology*, vol. 30, no. 5-6, pp. 497–506, 2006.
[19] A. Krontiris and K. E. Bekris, "Dealing with difficult instances of object rearrangement," in *Proceedings of Robotics: Science and Systems (RSS)*, 2015.
[20] A. Lambert, "Disassembly sequencing: A survey," *International Journal of Production Research*, vol. 41, pp. 3721–3759, 11 2003.
[21] B. Lazzerini and F. Marcelloni, "A genetic algorithm for generating optimal assembly plans," *Artificial Intelligence in Engineering*, vol. 14, no. 4, pp. 319–329, 2000.
[22] H. Ma, D. Harabor, P. J. Stuckey, J. Li, and S. Koenig, "Searching with consistent prioritization for multi-agent path finding," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, no. 1, 2019, pp. 7643–7650.
[23] T. Pan, A. M. Wells, R. Shome, and L. E. Kavraki, "A general task and motion planning framework for multiple manipulators," in *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2021, pp. 3168–3174.
[24] M. Phillips and M. Likhachev, "SIPP: Safe interval path planning for dynamic environments," in *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, 2011, pp. 5628–5635.
[25] H. Poschmann, H. Brüggemann, and D. Goldmann, "Fostering end-of-life utilization by information-driven robotic disassembly," *Procedia CIRP*, vol. 98, pp. 282–287, 2021.
[26] G. Sharon, R. Stern, A. Felner, and N. R. Sturtevant, "Conflict-based search for optimal multi-agent pathfinding," *Artificial Intelligence*, vol. 219, pp. 40–66, 2015.
[27] R. Shome, K. Solovey, A. Dobson, D. Halperin, and K. E. Bekris, "dRRT*: Scalable and informed asymptotically-optimal multi-robot motion planning," *Autonomous Robots*, vol. 44, no. 3–4, pp. 443–467, 2020.
[28] T. Siméon, J.-P. Laumond, J. Cortés, and A. Sahbani, "Manipulation planning with probabilistic roadmaps," *The International Journal of Robotics Research*, vol. 23, no. 7-8, pp. 729–746, 2004.
[29] A. Sintov and A. Shapiro, "Time-based RRT algorithm for rendezvous planning of two dynamic systems," in *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, 2014, pp. 6745–6750.
[30] S. Smith, G. Smith, and W.-H. Chen, "Disassembly sequence structure graphs: An optimal approach for multiple-target selective disassembly sequence planning," *Advanced Engineering Informatics*, vol. 26, no. 2, pp. 306–316, 2012.
[31] K. Solovey, O. Salzman, and D. Halperin, "Finding a needle in an exponential haystack: Discrete RRT for exploration of implicit roadmaps in multi-robot motion planning," *The International Journal of Robotics Research*, vol. 35, no. 5, pp. 501–513, 2016.
[32] S. Srivastava, E. Fang, L. Riano, R. Chitnis, S. Russell, and P. Abbeel, "Combined task and motion planning through an extensible planner-independent interface layer," in *2014 IEEE International Conference on Robotics and Automation (ICRA)*, 2014, pp. 639–646.
[33] I. A. Şucan, M. Moll, and L. E. Kavraki, "The Open Motion Planning Library," *IEEE Robotics & Automation Magazine*, vol. 19, no. 4, pp. 72–82, December 2012.
[34] Y. Tian, J. Xu, Y. Li, J. Luo, S. Sueda, H. Li, K. D. D. Willis, and W. Matusik, "Assemble them all: Physics-based planning for generalizable assembly by disassembly," *ACM Transactions on Graphics*, vol. 41, no. 6, 2022.
[35] M. Toussaint, "Logic-geometric programming: An optimization-based approach to combined task and motion planning," in *Proceedings of the 24th International Conference on Artificial Intelligence*, ser. IJCAI'15. AAAI Press, 2015, pp. 1930–1936.
[36] ——, "A tutorial on Newton methods for constrained trajectory optimization and relations to SLAM, Gaussian process smoothing, optimal control, and probabilistic inference," in *Geometric and Numerical Foundations of Movements*, ser. Springer Tracts in Advanced Robotics. Springer, 2017, vol. 117, pp. 361–392.
[37] M. Toussaint and M. Lopes, "Multi-bound tree search for logic-geometric programming in cooperative manipulation domains," in *2017 IEEE International Conference on Robotics and Automation (ICRA)*, 2017, pp. 4044–4051.
[38] M. Toussaint, K. R. Allen, K. A. Smith, and J. B. Tenenbaum, "Differentiable physics and stable modes for tool-use and manipulation planning," in *Proceedings of Robotics: Science and Systems (RSS)*, 2018.
[39] J. van den Berg and M. H. Overmars, "Roadmap-based motion planning in dynamic environments," *IEEE Transactions on Robotics*, vol. 21, no. 5, pp. 885–897, 2005.
[40] ——, "Planning time-minimal safe paths amidst unpredictably moving obstacles," *The International Journal of Robotics Research*, vol. 27, no. 11-12, pp. 1274–1294, 2008.
[41] W. Vega-Brown and N. Roy, "Asymptotically optimal planning under piecewise-analytic constraints," in *Algorithmic Foundations of Robotics XII (WAFR 2016)*, ser. Springer Proceedings in Advanced Robotics. Springer, 2020, vol. 13, pp. 528–543.
[42] G. Wagner and H. Choset, "M*: A complete multirobot path planning algorithm with performance bounds," in *2011 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2011, pp. 3260–3267.
[43] J. Yu and S. M. LaValle, "Structure and intractability of optimal multi-robot path planning on graphs," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 27, no. 1, 2013, pp. 1443–1449.