A General and Flexible Search Framework for Disassembly Planning

Timothy Ebinger¹,*, Sascha Kaden²,*, Shawna Thomas¹, Robert Andre², Nancy M. Amato¹, and Ulrike Thomas²

Abstract—We present a new general framework for disassembly sequence planning. This framework is versatile allowing different types of search schemes (exhaustive vs. preemptive), various part separation techniques, and the ability to group parts, or not, into subassemblies to improve the solution efficiency and parallelism. This enables a truly hierarchical approach to disassembly sequence planning.

We demonstrate two different search strategies using this framework that can either yield a single solution quickly or provide a spectrum of solutions from which an optimal may be selected. We also develop a method for subassembly identification based on collision information. Our results show improved performance over an iterative motion planning based method for finding a single solution and greater functionality through hierarchical planning and optimal solution search.

I. INTRODUCTION

Disassembly Sequence Planning (DSP) identifies physically viable plans to disassemble an assembly of parts. DSP is often used in end-of-life product design to verify the future ability to disassemble the product for recycling or repairs [1]. DSP can also be applied to Assembly Sequence Planning (ASP) using the concept of assembly-by-disassembly [2] where disassembly sequences can be reversed to assemble the product. This field is crucial towards a more automated product design and manufacturing process.

Existing methods can solve challenging DSP and ASP problems. They typically plan the translational case only [3] or apply directional blocking graphs [4]. They usually do not explore different search models (e.g., exhaustive vs. preemptive), employ a suite of part separation techniques, or optimize for different metrics within the same method. This gives need to a standard, flexible framework for searching the DSP/ASP solution space in a possibly hierarchical way. With such a framework, one could either search for a single solution as quickly as possible (e.g., for demonstrating disassembly viability in rapid prototyping), the solution that maximizes simultaneous part removal, the solution that performs the least complex motions (e.g., for easier control of robotic arms that will manipulate the assembly), or the solution that maximizes parallelism by favoring large, equal-sized subassemblies in a divide-and-conquer approach. This opens up a whole new way of approaching DSP/ASP.

We address these issues by developing a framework to search the disassembly sequence space in a variety of ways. Our framework is general and flexible: it is not limited to any single search style, separation approach, optimization metric, or termination criteria. It uses a directed acyclic graph called the disassembly graph to represent configurations (or placements of parts) and describes the transition from one configuration to another. Thus, DSP may be solved by constructing a disassembly graph where the root represents the assembled product and the leaves correspond to disassembled configurations and then extracting a path from the root to one of the leaves. We also develop a technique for identifying subassemblies that may be removed together and integrate it into the framework to produce hierarchical strategies.

We demonstrate the framework’s generality by providing two sample instantiations: a preemptive depth-first-search (DFS) that stops as soon as any solution is found and a full breadth-first-search (BFS) that can be used to find optimal paths by some given metric. We show how the framework can use a suite of separation techniques (e.g., mating vectors [5], sampling-based motion planning [6]).

We examine a variety of 3D problems, from puzzles to real world assemblies of up to 32 parts. We show improved performance over an iterative motion planning based method and in many cases is the only method able to solve the problem. We demonstrate the variety of disassembly sequences achieved even for relatively simple problems. Finally, we investigate the impact of subassemblies in speeding up disassembly by allowing them to be dismantled concurrently.

II. PRELIMINARIES AND PREVIOUS WORK

Let $A = \{a_1, a_2, ..., a_n\}$ be a set of parts, where each part has its own geometry and position in the assembly. DSP searches for viable disassembly sequences to separate $A$. Solutions give both the high-level ordering of parts (or groups of parts) for removal as well as collision-free removal trajectories. The high-level ordering is often called the precedence relation. A successful disassembly occurs when every $a_i \in A$ has been moved a sufficient distance away from all the others. We next discuss several different types of DSP approaches including mating vectors, sampling-based motion planning, and learning-based methods.

A. Separation Approaches

The simplest method is the mating vector approach [5]. Mating vectors identify a set of directions to attempt removals along, often either surface normals of part faces or a set of precomputed random directions. Parts are separated by
translating them along one of these directions until successful or a collision occurs. Directions are exhaustively searched.

The advantage of this approach is that separation paths are simple and computation is fast. However, it fails if complex paths are required such as those using multiple translation directions, (i.e., $k$-directional disassembly planning, see Fig. 1a), rotations, or if the assembly is nested demanding coordinated part removal (see Fig. 1b).

Other separation approaches include geometry analysis of best removal directions and identification of simultaneous part removal [7] and physics-based simulations [8].

### B. Sampling-based Motion Planning Approaches

Another class of techniques uses sampling-based motion planning (SBMP) to solve disassemblies requiring complex removal trajectories. Motion planning finds a path between two given object configurations (placements). SBMP techniques construct a graph [9] or tree [6] in configuration space, the set of all possible placements (valid or not) [10]. This structure is called the roadmap where nodes represent valid configurations and edges represent feasible trajectories between them. This roadmap may be queried to find solution paths, SBMP may be tailored to different applications by changing the definition of the moveable object, what placements are allowed, and how search is conducted.

SBMP can be applied to DSP by considering multiple movable objects, allowing only collision-free placements and trajectories, and building the roadmap until it contains a node where the parts are sufficiently far apart [11]. Multiple trees may be used in an RRT-style planner [12].

Many methods focus on removing a single part, as needed in product repair applications. Targetless RRT (T-RRT) [13] iteratively removes exterior parts using an approach similar to mating vectors. Cost optimization in the context of T-RRT planning has been explored [14]. D-plan [15] uses a retraction method to generate samples in narrow passages and constrained motion interpolation to connect samples together. As these methods focus on single part extraction, they do not efficiently extend to full disassembly problems.

Often methods struggle when there are subassemblies that must be moved together to find a solution. Most methods either ignore planning subassemblies or handle them implicitly by allowing the movement of one part to produce a responsive movement in another. An iterative planner (I-ML-RRT) [16] based on ML-RRT [17] classifies parts as either active and passive during a separation attempt. Explicit RRT planning is only done for the active part, but if it collides with any passive parts, then a subset of the passive parts will be perturbed in an effort to clear removal paths. Even though only single parts are moved, this allowed perturbation of other parts upon collisions enables this planner to solve some problems requiring coordinated part motions.

### C. Learning-based Methods and Optimality

Many learning-based methods have been applied to DSP including petri nets [18], [19], genetic algorithms [20], [21], [22], particle swarm optimization [23], and reinforcement learning [24]. These are typically not evaluated on realistic assemblies, do not consider subassemblies, and tend to focus on evaluating the feasibility of applying these techniques to DSP. Optimal planning, where the shortest sequence possible is required, has been studied in [18], [24], [25].

### III. Disassembly Planning Framework

We have developed a general disassembly planning framework to search the disassembly sequence space in a variety of ways. This framework is inspired by the SBMP framework [9], [6]. Just as SBMP uses a roadmap to encode the search progress, we will use a disassembly graph to organize the high-level removal orderings explored. A secondary roadmap data structure will store the low-level removal trajectories.

Alg. 1 gives the framework’s workflow. It searches by iteratively selecting a leaf from the disassembly graph $DG$ to explore that has not been completely disassembled (SELECTNODE) until some search criteria is met (DONE). The selected leaf contains a list of parts and/or subassemblies that still need to be separated. It selects a part or subassembly from this list and attempts removal (EXPAND). If successful, it adds the new disassembly graph node to $DG$ and the removal trajectory to the roadmap $R$. Once the search completes, it extracts a path from the root of $DG$ (i.e., the initial assembly) to one of its fully disassembled leaves. If multiple paths exist, it selects the one that minimizes some user-defined path metric. This path produces a high-level removal sequence and identifies a corresponding path in $R$ that provides the specific removal trajectories.

Note that the framework fully integrates subassembly identification into the search process. By allowing disassembly graph nodes to store parts and subassemblies, search may proceed in a hierarchical fashion. We present a subassembly identification technique later in Section IV.

We next describe the relationship between the disassembly graph and roadmap data structures and discuss how framework components may be implemented to achieve different search behaviors and solution complexities.

### A. Framework Data Structures

We use two correlating data structures: the disassembly graph $DG$ and the roadmap $R$. $DG$ serves as an abstract overview the search where each node represents a different state of the assembly (i.e., which parts have been removed and which ones belong to subassemblies). It is initialized
Algorithm 1 General Disassembly Planning Framework

1: function Disassemble(A)
2:     Input: A list of parts A = \{a_1, a_2, ..., a_n\}
3:     Output: A disassembly sequence seq and removal trajectory set T
4:     Initialize DG to contain a single root node where A is assembled
5:     Initialize R to contain a single configuration where each part in A is in its original position
6:     while !DONE(DG) do
7:         d ← SELECTNODE(DG)
8:         nodes, paths ← EXPAND(DG, R, d)
9:         if nodes ≠ ∅ then
10:             add nodes to DG and paths to R
11:         end if
12:     end while
13:     return seq, T ← EXTRACTPATH(DG, R)
14: end function

with a single node representing the initial assembly. DG is a directed acyclic graph where edges indicate that a removal path exists to take the assembly from one state to the other. Removal paths are stored in a separate roadmap data structure R. In R, nodes represent assembly configurations (i.e., specific placements of each part in space) and edges indicate that a collision-free trajectory exists between two configurations. This is the same data structure as used in SBMP. Fig. 2 provides an example of a disassembly graph and corresponding roadmap configurations. It contains:

1) A root node with the completely assembled model (parts A, B and C) inside a container.
2) Separation of the multi-part subassembly A and B, marked by parentheses; these parts must still be separated for complete disassembly.
3) Separation of part C from the container.
4) Disassembly of the remaining two parts A and B.

Fig. 2: An example disassembly graph (left) and the roadmap configurations each node represents (right). Parts in a disassembly graph node grouped in parentheses indicate a subassembly that was removed together and disassembled separately.

B. Realizing Different Search Strategies

The framework can be used to implement a variety of search patterns by simply adjusting how nodes are selected from the disassembly graph for exploration (SELECTNODE), how they are explored (EXPAND), and the termination criteria (DONE). We discuss search patterns for solving two classes of disassembly problems: finding any solution as quickly as possible and finding a solution that optimizes a given metric. Finally, we discuss how subassemblies impact search cost.

1) Searching for Any Viable Solution: When searching for any viable solution, computation time is prioritized. Preemptive depth-first search is a good candidate for this type of application as it explores the space until a single solution is found. Faster, simpler separation approaches are favored over more expensive, complex ones. Likewise, single part removals are attempted before subassemblies are explored. It selects the most recent unexplored leaf to explore and terminates as soon as a fully disassembled leaf is found.

2) Searching for an Optimal Solution: One way to find a solution that optimizes some metric is to fully enumerate all possibilities and report an optimal one. A full breadth-first-search of the space is a natural choice. Other benefits of a full enumeration include characterizing the solution space and comparing optimal solutions for different metrics. The framework can explore the disassembly sequence space this way by selecting the highest-level disassembly graph leaf that has not fully exhausted the available separation methods. Search is terminated when no such leaf exists. Of course, one could find the optimal solution without a full enumeration by instead preforming an A*-style search that terminates as soon as an optimal path is found.

3) Impact of Subassemblies on Search: The disassembly search space is very large: given n parts, there are up to n! different disassembly sequences possible when only moving single parts at a time. While not all disassembly sequences are viable (i.e., requires parts to collide or pass through one another), these sequences are not discovered until attempted. Thus, searching in a breadth-first way to find the sequences that optimize some metric can become quite costly.

Subassemblies can help reduce the search space if they define constraints on possible solutions. This is frequently the case in manufacturing where subassemblies are constructed first and then put together into the final product. In that situation, subassembly membership is provided as input. Consider the impact of enforcing only a single subassembly of size s. The number of possible (although not necessarily viable) disassembly sequences reduces from n! to s! + (n − s + 1)!, the number of sequences for the subassembly plus the number of possibilities for the remaining parts considering the subassembly as a single part. When all possible subassemblies are allowed without restriction, they significantly increase the search space. It has been shown that the number of cuts for an assembly (i.e., the number of possible subassemblies) increases exponentially with the number of parts [3]. Thus it is critical that the search space is pruned effectively.

C. Expansion Approaches for Part Separation

The framework allows any type of separation method as long as that method provides removal trajectories in addition to identifying if a part (or group of parts) can be removed.
Separations are attempted in the EXPAND function of Alg. 1. Alg. 2 outlines the general EXPAND function.

Algorithm 2 Attempting Separation

1: function EXPAND(DG, R, d)
2: Input: A disassembly graph DG, a roadmap R, and a disassembly node d to attempt separation on
3: Output: A set of 0 or more disassembly graph nodes N and possible removal trajectories P
4: subs = IDENTIFYSUBASSEMBLIES(dremain) where dremain is the list of parts in d not yet separated
5: Select s from subs to attempt separation
6: Select a separation method m from M
7: (node, path) = m.SEPARATE(d, s)
8: Add node to N and path to P
9: return (N, P)
10: end function

Given a disassembly graph node d (that contains the list of parts and/or subassemblies that still need to be separated), EXPAND may first check if any parts may be grouped into subassemblies that are not already grouped. IDENTIFYSUBASSEMBLIES returns a list of items to be separated, either single parts, subassemblies, or combinations thereof. (Subassembly identification is discussed in detail in Section IV.) EXPAND selects an item from this list to attempt removal. It can either make a single attempt or try all available separation methods (stored in set M), depending on the desired search pattern and application. For simplicity, Alg. 2 demonstrates a single attempt. Multiple attempts may be easily implemented by repeating lines 6 and 7 with different separation methods.

If successful, EXPAND adds a new disassembly graph node to DG as a child of d and adds the removal trajectory to R as a series of nodes (configurations) and edges following the roadmap node corresponding to d in DG. If not successful, the attempt is noted to avoid future reattempts.

The results in this paper use two different separation methods: mating vectors [5] and SBMP via an RRT exploration [6]. We discuss each in turn below.

1) Mating Vector Approach: Recall that the mating vector approach separates parts (or subassemblies) by linear translations along a single direction, stopping when the part has collided with another part or obstacle or when the part has achieved a sufficient separation distance. In the results for this paper, we define the set of mating vectors from the surface normals of all parts in the assembly, filtering out directions that are too similar as given by a user-defined threshold. We try all of these directions whenever attempting separation, stopping at the first successful one. The removal trajectory is simply a pair of nodes (the part placements before and after this separation) with an edge connecting them. As edges in the roadmap represent linear interpolations of the endpoints, this fully describes the removal trajectory.

2) SBMP Approach: Given a node d from DG with its associated state of the assembly as a configuration c and a subassembly (single- or multi-part) to remove, we grow a targetless RRT-style tree using c as the tree’s root. When growing the tree, we only sample the degrees of freedom for the subassembly in question leaving all other parts and obstacles in the environment static. We allow the tree to grow until it has successfully achieved separation or hits a limit in number of attempts or time. This prevents the method from thrashing in a highly constrained region. If it is successful, all configurations along the separation will be returned as path for storage in R, and a single new disassembly node node representing the final state of the RRT-style removal will be added to DG.

Recall that the standard RRT method extends a tree node in a random direction qrand until a collision occurs or a maximum distance is reached. While this achieves greater travel distances and hopefully final separation, it struggles in constrained settings (e.g., inside a container with a single opening) where moving items to the maximum allowed does not make progress toward final separation (e.g., when it needs to travel through an opening in the container), such as the situation in Fig. 1a. To mitigate this, we adjust the sampling of qrand to be near existing tree samples. This helps prevent the RRT from thrashing against container walls when trying to remove a part through a small hole on the border of a relatively spacious container.

IV. Subassembly Identification

Often, subassemblies are known ahead of time as part of the design and manufacturing process. When they are not, we can exploit collision information obtained from mating vectors [5] to identify prospective subassemblies. This significantly prunes the subassembly search space.

Recall that mating vectors represent a reduced set of separation directions (e.g., face normals). To identify subassemblies, we first examine each part and record the mating vectors that do not result in collision with a static obstacle such as a fixed container. We call these feasible directions. For each feasible direction for a given part, we then record which parts it collides with upon the first collision when translating along that direction. Parts that collide first represent candidates for subassembly grouping.

We build subassemblies as follows. For a given part a, we examine each of its feasible directions dir. If translating along dir led to a collision with another part b, then we check that dir is feasible for b as well and add it to the current candidate subassembly. We repeat this process of checking dir’s feasibility, adding parts as before, until we run out of colliding parts along dir or dir becomes unfeasible (i.e., collision with a static obstacle). If dir is found unfeasible, we discard the subassembly candidate. We also discard any subassembly duplicates. Thus for each part and direction, a maximally-sized subassembly is formed based on these rules.

Fig. 1b shows a simple assembly, with two subassemblies: \{A, B, C\} and \{A, B, C, D\}, see the dashed lines. The first subassembly is created by examining part A as follows:

- The top direction is discarded because it collides with a static obstacle (the container).
Analyzing not explicitly identify subassemblies but does allow the
We compare this approach against I-ML-RRT which does
subassembly identification from mating vector directions.

B. Preemptive DFS Performance

Puzzle 2D performs with and without rotations.
single part rotations. We also explore how the Triple Stacked
method and I-ML-RRT consider search spaces including
rotation and translation of the two large pieces, both our
As the Simplified Coaxial Connector requires a simultaneous
search space when translations are known to be sufficient.

Coaxial Connector may be solved with translations, we
receive joining pegs to hold the assembly together.
example, the Frame has 8 slot nuts embedded in the rods to
emphasize the need for explicit subassembly identification.

puzzle is the same as in [11]. The constrained box with pegs
was also studied in [16] and the pentomino
samples, see Fig. 3. Some models are constrained by placing
54x620}.

All methods where implemented in C++ and compiled
using GCC 7.1.1. All experiments were run on CentOS with
an Intel Core i7-3770 and 16 GB of memory.

A. Benchmark Models

We study a variety of assemblies including toy problems
to stress different parts of the algorithms and real world ex-
amples, see Fig. 3. Some models are constrained by placing
them inside a static box with a single opening. The triple
planar puzzle was also studied in [16] and the pentomino
puzzle is the same as in [11]. The constrained box with pegs
emphasizes the need for explicit subassembly identification.
Note that some assemblies have very tightly fitting parts. For
example, the Frame has 8 slot nuts embedded in the rods to
receive joining pegs to hold the assembly together.

As all of these disassembly problems except the Simplified
Coaxial Connector may be solved with translations, we
limit exploration to the translational degrees of freedom.
This is often done in most previous work to restrict the
search space when translations are known to be sufficient.
As the Simplified Coaxial Connector requires a simultaneous
rotation and translation of the two large pieces, both our
method and I-ML-RRT consider search spaces including
single part rotations. We also explore how the Triple Stacked
Puzzle 2D performs with and without rotations.

B. Preemptive DFS Performance

The first realization of the framework we demonstrate is
a preemptive DFS search. Here the goal is to find a valid
disassembly sequence as quickly as possible. We also employ
subassembly identification from mating vector directions.
We compare this approach against I-ML-RRT which does
not explicitly identify subassemblies but does allow the
movement of single parts to elicit a responsive movement
in others. Our results show that our method produces shorter
paths in less time.

It should be noted that our implementation of I-ML-RRT
follows [16] as closely as possible given the information in
that work, but some differences remain. In [16], a Gaussian
distribution is used and dynamically updated to bias the
random selection towards parts that have been found as

easier to move. However, not enough details are provided
for accurate implementation, so we use purely random part
selection when choosing a part to perturb, and iterate se-
quently through the list of remaining parts when choosing
an active part. The list ordering was found in [16] to have
little impact on run time, when not intentionally chosen as
the worst ordering. We limit each call of RRT based on time
and node count (whichever is reached first) as the ideal node
count of 200 can be very slowly reached in some examples.
The average number of nodes for a single ML-RRT iteration
was just under 80 for the Triple Stacked Puzzle 2D and
had similar results to the 200 node limit in [16]. Similarly,
we also limit the maximum number of part perturbation
iterations to 20 (i.e., for each failed sample of the active part,
up to 20 passive parts can be moved). This prevents initial
thrashing of the method without loss of overall performance.

1) Framework Specialization: There are many ways to
realize a preemptive DFS with the framework. For the results
in this paper, we make the following choices in Alg. 1:

- **DONE** returns true as soon as a single disassembly node
  is found with all parts fully disassembled.
- **SELECTNODE** selects the last viable disassembly node
created in **DG** for further exploration.
- **EXPAND** iterates through the following set of separation
  methods in increasing complexity until successful.
  1) Mating vector separation for all remaining single
     parts (see Section III-C.1)
  2) Mating vector separation for all remaining sub-
     assemblies (see Section IV)
  3) SBMP separation via RRT for all remaining single
     parts (see Section III-C.2)
  4) SBMP separation via RRT for all remaining sub-
     assemblies

If none are successful, items 3 and 4 are continuously
tried until successful removal, some progress is made, or
a timeout occurs and the strategy fails. Progress can be
made without a removal if an RRT sample of clearance
higher than the starting configuration has been added to
the RRT’s roadmap, as is also done in [16]. Upon any
progress, the iteration restarts to item 1.

2) Performance Comparison: Table I compares the suc-
cess rate, average running time, and solution path metrics
for our approach and I-ML-RRT. Data is averaged over 10
runs with a timeout of 5000 seconds, except for the Gearbox
and Engine examples which were given 10000 and 15000
seconds, respectively. Average times and path metrics are
only reported for successful runs.

The Preemptive DFS method is the only one to always
successfully solve each problem within the allotted time. It is also at least an order of magnitude faster than I-ML-RRT. The improved performance may be attributed to both the use of mating vectors and subassemblies. Mating vector exploration is faster than RRT exploration, even if the RRT exploration was constrained to mating vector directions. The more frequently mating vector separation is successful, the greater the improvement in running time. Recall that I-ML-RRT does not handle subassemblies and only allows for adjusting other parts if the single part being attempted causes collisions. This is a slow way to handle problems that require coordinated motion of parts, and means it cannot solve problems that truly require \textit{simultaneous} movements of parts, like our Constrained Block and Pegs example where there are tight-fitting parts of the assembly (e.g., the rod between the two blocks, all surrounded by a snug container).

Preemptive DFS produces much shorter paths in terms of numbers of roadmap nodes due to its use of mating vectors in addition to RRT exploration for all problems. The number of roadmap nodes along the path indicates the complexity of movement required as nodes occur at direction changes. Here we see the impact of mating vector separation on path simplicity.

It should be noted that the times reported for similar puzzles in [16] are not the same. For the Triple Stacked Puzzle (called Planar Puzzle in that work), the difference is roughly an order of magnitude. For the Pentomino puzzles, it is unclear whether the reported times are with or without the constraining box in [16]. There are some implementation differences that our version has which could contribute to time disparities, as mentioned earlier in this section. It is also important to note that we replicated the examples to the best of our ability, but they are not the identical puzzles used. The initial clearance of parts in the puzzles might be lower due to tighter obstacles, which is identified in [16] to rapidly hurt solving times for the method.

### C. Solution Heterogeneity via Full BFS

We next use the framework to implement a full BFS approach. This approach is useful when an optimal solution respecting some metric other than runtime is needed, or to study the types of possible solutions available. Here we use it to explore the latter. Other work in DSP to date currently stops once a solution is found, and thus is not comparable.

1) \textbf{Framework Specialization:} There are many ways to realize a full BFS with the framework. For the results in this paper, we make the following choices in Alg. 1:

- \textit{DONE} returns true when all nodes in the disassembly graph have been fully explored.
- \textit{SELECTNODE} selects the closest disassembly node to the root that has not been explored.
- \textit{EXPAND} iterates through the following set of separation methods for each single part and then for each subassembly (see Section IV) until successful or all methods are exhausted:

1) Mating vector separation (see Section III-C.1)
2) SBMP separation via RRT (see Section III-C.2)

2) \textbf{Demonstration:} We demonstrate Full BFS on a Single Stacked Puzzle (a 6-part subset of the Triple Stacked Puzzle in Fig. 3a) and the Constrained Block and Pegs Assembly (Fig. 3d) as these have few enough parts, and obstacles to reduce the number of attempted subassemblies, to fully explore the solution space in a reasonable amount of time.

\begin{table}[h]
\centering
\caption{Performance of Full BFS.}
\begin{tabular}{|c|c|c|c|}
\hline
Model & Time (sec) & \# of Valid Solutions & \textit{Path Length Range} (nodes) \\
\hline
Single Stacked Puzzle & 79.81 & 5446 & 12 – 48 \\
Constrained Block/Pegs & 550.36 & 560 & 28 – 97 \\
\hline
\end{tabular}
\end{table}

The full search identified hundreds to thousands of collision-free disassembly solutions for both problems, each with unique disassembly sequences. This of course comes at the cost of much larger running times, which is exponential
with the number of parts. Even for these small problems, there is a wide range of path lengths in the solution set. The Full BFS strategy finds a solution with shorter path length than either the Preemptive DFS or I-ML-RRT which both terminate after finding the first viable solution, see Table I.

Not only does this strategy find shorter paths, it can probe the distribution of solution path lengths. Fig. 4 shows the path length histograms for both problems. The Single Stacked Puzzle is a simple problem where subassemblies are not required and parts may be removed using only single translations toward the opening, favoring the mating vector approach. This is reflected in the path length distribution where it has a single peak at a relatively short path length.

The Constrained Block and Pegs problem has a different distribution (multiple peaks, less heavily weighted toward shorter/mating vector paths). Unlike the prior problem, it requires subassemblies and more complex extraction paths (using RRT) to properly position the parts near the opening.

The search space can also be reduced by being more selective when identifying subassemblies. The method described in Section IV finds many subassemblies, particularly in the absence of static obstacles. Either pre-defining or otherwise reducing the number of attempted subassemblies would have a huge impact on the search space size.

### D. Gains by Parallel Disassembly through Subassemblies

The last exercise of the framework is to study how subassemblies impact the ability to virtually parallelize disassembly. This is useful in situations where either the computation or the actual manipulation may be parallelized by working on independent subassemblies. This exploits the natural parallelization of DSP: when one subassembly is removed, it can be independently disassembled.

1) Framework Specialization: Here we use the Preemptive DFS specialization given in Section V-B.1 for demonstration. However, instead of automatically identifying subassemblies, they are predefined by the user to maximize their size. Multiple threads are not used in the implementation, but rather multiple timers measure how long it takes for the initial subassembly removals and then the maximum time to disassemble any of the resulting subassemblies.

2) Performance Comparison: Table III compares three different virtual running times:

- “Parallel” — the time to remove subassemblies plus the maximum time to dismantle any of the remaining subassemblies.
- “Sequential” — the time taken with the dismantling of subassemblies serialized.
- “Without Subassemblies” — the time taken by Preemptive DFS when subassemblies are not allowed.

![Path length histograms of the full search space.](image)

Here we used the framework to fully explore the solution space to show that even for small problems a wide variety of solutions exist. This could be used to find paths that optimize some metric, but a more efficient approach would be to perform an A* style search and terminate as soon as an optimal path is found. The framework can easily support such a search by changing `SELECTNODE` to instead return the node that optimizes the given heuristic and setting `DONE` to terminate as soon as a solution is found.

---

### TABLE I: Comparison of Preemptive DFS to I-ML-RRT for the models in Fig. 3. Results are averaged over 10 runs, excluding failed runs. All methods are restricted to translation except for the Simplified Coaxial Connector (which requires rotation) and the rotation version of the Triple Stacked Puzzle 2D.

<table>
<thead>
<tr>
<th>Model</th>
<th>Success Rate (%)</th>
<th>Avg. Successful Time (sec)</th>
<th>Avg. Path Length (nodes)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Preemptive DFS</td>
<td>I-ML-RRT</td>
<td>Preemptive DFS</td>
</tr>
<tr>
<td>Triple Stacked Puzzle 2D Translation</td>
<td>100</td>
<td>100</td>
<td>0.30</td>
</tr>
<tr>
<td>Pentomino</td>
<td>100</td>
<td>100</td>
<td>0.05</td>
</tr>
<tr>
<td>Constrained Pentomino</td>
<td>100</td>
<td>100</td>
<td>0.11</td>
</tr>
<tr>
<td>Constrained Block and Pegs</td>
<td>100</td>
<td>0</td>
<td>19.57</td>
</tr>
<tr>
<td>Frame</td>
<td>100</td>
<td>0</td>
<td>23.27</td>
</tr>
<tr>
<td>Toy Plane</td>
<td>100</td>
<td>90</td>
<td>6.46</td>
</tr>
<tr>
<td>Drill</td>
<td>100</td>
<td>0</td>
<td>25.81</td>
</tr>
<tr>
<td>Gear</td>
<td>100</td>
<td>0</td>
<td>208.70</td>
</tr>
<tr>
<td>Gearbox</td>
<td>100</td>
<td>0</td>
<td>5352.27</td>
</tr>
<tr>
<td>Engine</td>
<td>100</td>
<td>0</td>
<td>10638.08</td>
</tr>
<tr>
<td>Simplified Coaxial Connector</td>
<td>100</td>
<td>100</td>
<td>50.98</td>
</tr>
<tr>
<td>Triple Stacked Puzzle 2D Rotation</td>
<td>100</td>
<td>100</td>
<td>0.31</td>
</tr>
</tbody>
</table>
using automatically identified subassemblies, even when the automatically identified subassemblies give greater solution flexibility. Often the savings are significant as in the case of the Frame, Drill, and Gear. Using predefined subassemblies in a sequential fashion is beneficial over restricting the problem to single part removals for all problems except the Triple Stacked Puzzle in 2D. This is possibly due to the independent nature of the puzzle’s structure and does not as easily benefit from a subassembly hierarchy.

VI. CONCLUSION

We presented a new general framework for searching the disassembly solution space in a truly hierarchical way. The framework is not limited to any single search style, part separation approach, optimization metric, or termination condition. We showcased the framework’s flexibility by demonstrating different types of search strategies on a variety of problems and employed different part separation philosophies. Our results show improved performance over I-ML-RRT. Finally, we studied the impact of subassemblies on search by exploring how the framework performs without subassemblies, with automatically identified subassemblies, and with predefined subassemblies.

REFERENCES


