Solving Rearrangement Puzzles
using Path Defragmentation in Factored State Spaces

Servet B. Bayraktar[1], Andreas Orthey[1,3], Zachary Kingston[2], Marc Toussaint[1], Lydia E. Kavraki[2]

Abstract—Rearrangement puzzles are variations of rearrangement problems in which the elements of a problem are potentially logically linked together. To efficiently solve such puzzles, we develop a motion planning approach based on a new state space that is logically *factored*, integrating the capabilities of the robot through factors of simultaneously manipulatable joints of an object. Based on this factored state space, we propose less-actions RRT (LA-RRT), a planner which optimizes for a low number of actions to solve a puzzle. At the core of our approach lies a new path defragmentation method, which rearranges and optimizes consecutive edges to minimize action cost. We solve six rearrangement scenarios with a Fetch robot, involving planar table puzzles and an escape room scenario. LA-RRT significantly outperforms the next best asymptotically-optimal planner by 4.01 to 6.58 times improvement in final action cost.

Index Terms—Task and Motion Planning, Constrained Motion Planning, Manipulation Planning

I. INTRODUCTION

REARRANGEMENT puzzles are difficult instances of rearrangement problems [17], [38], where objects are potentially logically linked to each other. Logically linked objects require the robot to first move one object, before a second can be moved. We define problems with this characteristic as rearrangement puzzles. Those problems are at the core of many robotics tasks like household chores or product assembly. For instance, a robot working in a production facility may need to rearrange objects on a production line, or a robot trapped inside a room has to find its way out (Fig. 1).

Rearrangement puzzles are often solved using one of three approaches. In a top-down approach, symbolic reasoning is used to guide exploration of the space [38], [10]. Such methods include most Task and Motion Planning (TAMP) solvers. TAMP solvers often start by computing action skeletons [6], which are then used to initialize lower-level motion planning [9] or optimization methods [37]. In a bottom-up approach, a search is conducted directly in the joint robot and object state space by carefully analyzing and sampling the constraints involved [23], [39]. A third class of methods
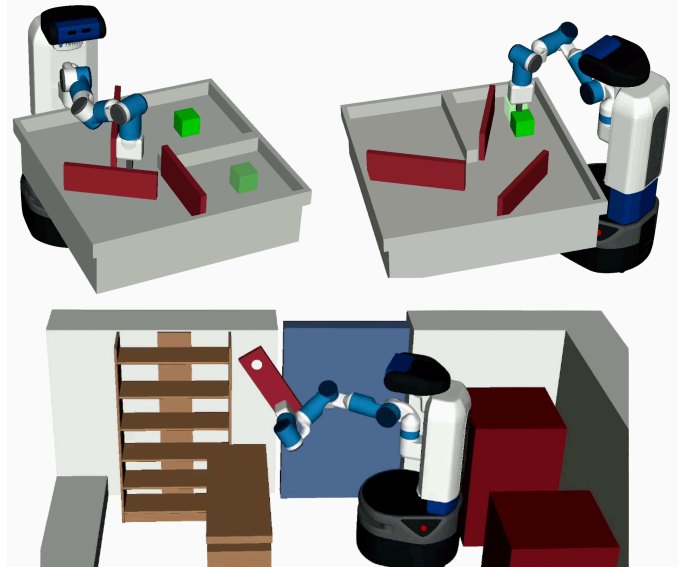
Figure 1: Example of a Fetch robot solving a puzzle with 3 doors, to move a green cube from a start to a goal configuration (Top), and a locked room scenario, where the Fetch robot has to escape by rearranging furniture and locks (Bottom).

uses integrated solvers, to selectively switch between lower and higher level abstractions. This can be achieved in different ways, for example using backtracking on failure [11], or by switching between joint-space sampling and sampling in region where constraint switches occur [29], [36]. Methods for rearrangement puzzles, however, often do not scale well, because they might require excessive backtracking [4] or lack good heuristics to guide a solver to a solution [36].

To tackle this issue, we propose a complementary method to compute an efficient lower bound on the solution, i.e. an *admissible heuristic* [16], [25]. This admissible heuristic is obtained by computing feasible paths for the objects alone, while ignoring the robot. We show that this admissible heuristic can be efficiently computed, and is able to act as a guide to solve the complete problem involving multiple manipulation actions of the robot.

However, computing such an admissible heuristic based on objects alone requires solving two problems. The first is the *capabilities problem*. If we ignore the robot and its capabilities, solution paths might not be executable by the robot, especially if multiple objects move at the same time—a one-armed robot might be incapable of manipulating two doors at the same time. The second is the *minimal actions problem*. Computing solutions for the objects alone might produce excessive pick-place sequences, which would be tedious and take too much

time for the robot to execute.

We propose two contributions to address these two issues. First, we propose a factored state space to solve the capabilities problem. This factored state space implicitly models the capabilities of the robot by grouping joints into *factors*, wherein joints in a factor are simultaneously manipulatable by a given robot. We develop novel interpolation and action cost functions to make this state space usable by general-purpose motion planners [19], [12]. Second, we propose a new method to significantly reduce the number of actions, which we call *path defragmentation*. Path defragmentation can reduce the number of actions by reasoning about the ordering of path segments through different factor spaces. This method is integrated into a new planner, the less-actions RRT (LA-RRT). LA-RRT is able to handle the non-additive property of our minimal-actions cost, and can split interpolated path segments into sequences of factored path segments, which are then optimized using path defragmentation. We apply LA-RRT to difficult instances of rearrangement puzzles where we assume that the mode of each object is constant and show that it can find paths with significantly less action cost compared to state-of-the-art planners. Eventually, we use those paths to compute complete manipulation sequences for a simulated Fetch robot. Fig. 2 shows an overview about our method.

II. RELATED WORK

We group approaches to rearrangement problems into three groups, namely top-down planning, bottom-up planning, and integrated planning approaches.

Top-down approaches are often based on computing *action skeletons*, sequences of symbolic actions which are then used as constraints for joint-space planning or optimization approaches [37], [11]. This approach is highly successful if the degrees of freedom in the environment are logically decoupled, like pick-place actions on distinct objects [17], [7], [8]. It is often sufficient in these scenarios to compute joint-space trajectories without feedback, even if multiple robots and large planning horizons are involved [38], [10]. However, if an action skeleton cannot be solved expensive backtracking is required [4] to validate another action skeleton.

Our work is complementary, in that we also execute symbolic actions on our robot, but we choose those actions *implicitly* using the (admissible) heuristic from our LA-RRT planner, which gives us a complete valid sequential solution path for all objects involved. This tighter integration of planning and robot capabilities give us a better chance to avoid expensive backtracking.

Another approach to rearrangement planning are bottom-up approaches. These approaches explicitly search through the combined space of the robot and objects [35]. This can be advantageous as sampling-based planners can be applied [15], which can achieve asymptotic optimality guarantees [39], [28], [27]. These methods often use a constraint-graph [22], [23], which enumerates the valid constraint combinations in a scene. Constraint-based methods require effective projection methods [2], [15] to sample and interpolate along constrained subspaces. While these approaches can provide strong guaran-

tees [39], scaling to higher dimensions (i.e., number of objects) adds significant challenges [14].

Our approach differs in that we do not plan directly in the full robot-object configuration space, which can be costly to compute. Instead, we plan first in the object-only configuration space by integrating the capabilities of the robot into the motion planner using the new factored state space (see Sec. III).

Finally, rearrangement problems can be solved with integrated methods. Those methods combine both planning on a high-level, like a symbolic layer, with planning on a lower level, like joint-space planning. A tight integration is crucial to avoid backtracking on infeasible high-level solutions. Hierarchical planners [1], [11] can often find good solutions with few backtracking operations. However, when objects are logically coupled, like in Navigation among moveable obstacles (NAMO) problems [30], it is often difficult to find the correct high-level solution. To tackle this issue, geometric information can often be integrated into the symbolic description [7], or sampling is extended to include joint configurations consistent with symbolic actions [29], [36]. While those approaches provide concise frameworks for optimal planning, they might suffer from slow convergence due to the high branching factor of possible actions.

To tackle the high branching factor, it becomes often necessary to find good heuristics. Recent approaches include reasoning about collision regions between objects [40], or by computing initial solutions, where every object is moved at most once (the monotone case) [41]. Our approach is complementary by also computing a heuristic. However, our heuristic is admissible, and is tailored to problems where objects are logically linked to each other.

The computation of this admissible heuristic is done using the new LA-RRT planner, which can efficiently reason over factored state spaces. This planner is inspired by planners like the Manhattan-like RRT (ML-RRT) [3]. In ML-RRT, planning is decomposed into an active and a passive subspace. Our planner LA-RRT, however, generalizes this idea to arbitrary subspaces (factors), makes it applicable to manipulation, and combines it with optimality, such that we approach the minimal number of switches between factor spaces.

III. FACTORED STATE SPACE

Our goal is to solve manipulation planning problems that require solutions through the combined state space of $X_R \times X$, where X_R is the robot's state space and X is the state space of the manipulatable objects. We decompose this problem by projecting this state space $X_R \times X \rightarrow X$ removing the robot state space. To account for the robot capabilities, we manually define instead a factored decomposition of X as $X = F_1 \times F_2 \times \ldots \times F_N$ as shown in Fig. 3, where a factor represents a subspace of X containing joints, which are *simultaneously manipulatable* by the robot. For convenience, we also define the operation k *in* F_i as returning the indices of the joints in F_i, and the operation $|F_i|$ as the number of joints.

Example: For a one-arm manipulator arm, a door with a single revolute joint (depicted as F_2 in red in Fig. 3), would have $|F_{door}| = 1$, while a movable planar disc would have $|F_{disc}| = 2$ (shown as F_1 in green in Fig. 3).
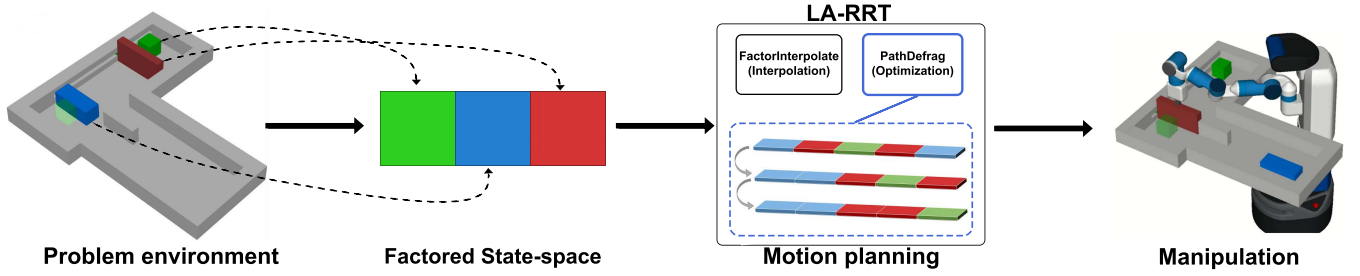
Figure 2: Overview about the system. Given a problem environment (**Left**), we model the objects through a factored state space (**Middle left**) where each object represents a factor in a different color. We then apply our algorithm LA-RRT to the factored state space, whereby we first exploit the factored interpolation, and then use a path defragmentation method to optimize the number of switches between factors in the resulting path (**Middle right**). This solution is then used to initialize a manipulator algorithm, which computes a complete manipulation sequence to solve the rearrangement puzzle (**Right**).
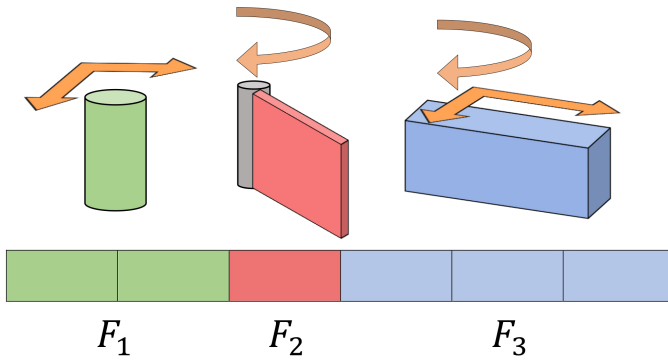

Figure 3: Example of the structure of the factors in the state-space. Each color represents a factor, and each joint of an object corresponds to a cell in the factor.

The purpose of the factors is to ensure that at most k objects move at the same time given k available manipulator arms. This constraint is an implicit way to define *action skeletons* [10], i.e., as a sequence of alternating factor-paths, which can then be executed individually by executing pick-place actions with the robot.

To plan in factored state spaces requires the implementation of three functionalities which are crucial for planning: interpolation, goal constraint, and cost.

Interpolation: A linear interpolation between two states would interpolate in all dimensions thereby moving objects simultaneously. To only move one factor at a time, we develop a Manhattan-like interpolation method applicable with arbitrary factor state spaces, which is depicted in Alg. 1. This method interpolates a path between two states x_{from} and x_{to} by using an interpolation parameter $t \in [0, 1]$, and an ordering of factor spaces[1]. As output, we return a state x_{out} at distance t between x_{from} and x_{to}. This methods works by first computing the individual distances between x_{from} and x_{to} when *projected* onto each factor (Line 1). We then find the factor space m where the interpolation variable t lies (Line 2). As an example, let us assume we have two factors with distances 1 and 3, and total distance 4. In the first step, we normalize them to $\frac{1}{4}$ and $\frac{3}{4}$. If $t < \frac{1}{4}$, we select the first factor. If $t \geq \frac{1}{4}$, we select the second

---

[1]In our evaluations, we use a random ordering, since we found no significant influence on performance for different orderings.

second factor.

All factor spaces before m are fully interpolated and set to the corresponding values of x_{to} (Line 4–6). Then, we call the intrinsic interpolation function of the selected factor F_m and change its corresponding indices in x_{out} (Line 8). Finally, we set the factors after m to the x_{from} values (Line 9–10), and return x_{out} (Line 11).

Algorithm 1: FACTORINTERPOLATE

Input: x_{from}, x_{to}, t, $F_1, \ldots, F_M$
Output: x_{out}

1   $d_{1:M} \longleftarrow$ GETDISTANCES$(x_{\text{from}}, x_{\text{to}}, F_{1:M})$
2   $m \longleftarrow$ FINDINDEX$(d_{1:M}, t)$
3   $d_{interpolated} \longleftarrow 0$
4   **for** $i = 1$ **to** $m - 1$ **do**
5   **foreach** j **in** F_i **do** $x_{\text{out}}[j] \longleftarrow x_{\text{to}}[j]$
6     $d_{interpolated} \mathrel{+}= d_i$
7   $s \longleftarrow (t - d_{interpolated}/d_m)$
8   $x_{\text{out}} \longleftarrow$ INTERPOLATE$(F_m, x_{\text{from}}, x_{\text{to}}, s)$
9   **for** $i = m + 1$ **to** N **do**
10   **foreach** j **in** F_i **do** $x_{\text{out}}[j] \longleftarrow x_{\text{from}}[j]$
11   **return** x_{out}

Goal: Next, we define a goal constraint by taking all joints into account which have a designated goal position. We call this set the goal-indices G_{I_G}. To take joint positions into account, we use a goal state x^g which is only defined for indices in G_{I_G}. All other indices are freely-chooseable and can be randomly sampled. For such a partial state x^g we represent an ϵ-*goal region* X_G as

$$X_G = \{x \in X \mid d(x_i, x_i^g) \leq \epsilon, i \in G_{I_G}\}. \quad (1)$$

Cost: Finally, we have to define a cost function, which reflects our desire to minimize the number of pick-place actions the robot has to execute. This can be done by counting the number of factor changes when interpolating between two states, i.e., $c_{\text{actions}} : X \times X \to \mathbb{N}_{\geq 0}$. However, this cost is difficult to define as we show in Fig. 4, because of two issues. First, we cannot discriminate between paths of different lengths (p_1 and p_3 in Fig. 4). We resolve this by adding a multi-layered cost function which works as c_{actions}, but behaves like a distance cost c_{dist} when the number of actions are equivalent. The second issue is
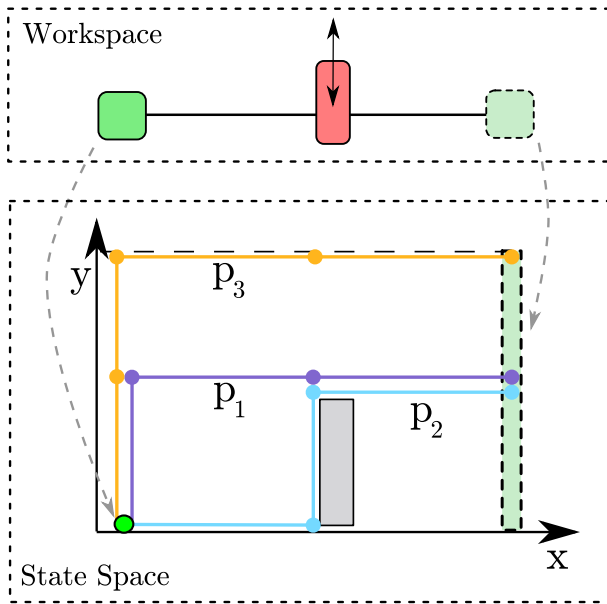
Figure 4: Explanatory example to demonstrate the difference between different cost functions. **Top**: Workspace of a single cube on a rail, which has to be moved from its start configuration (left) to a desired goal configuration (right). X-axis is the horizontal position of the green cube and the y-axis is the vertical position of the blocking cube. A blocking cube in the middle (red) needs to be moved out of the way to solve the problem. **Bottom**: The state space of this problem with the start configuration (green), goal region (lightgreen) and the collision-region (gray). We showcase three paths. p_1 is the optimal path with cost terms $c_{\text{actions}} = 2$, $c_{\text{additive}} = 3$, $c_{\text{dist}} = 3$, p_2 with $c_{\text{actions}} = 3$, $c_{\text{additive}} = 3$, $c_{\text{dist}} = 3$, and p_3 with $c_{\text{actions}} = 2$, $c_{\text{additive}} = 4$, $c_{\text{dist}} = 4$. This highlights two issues: (a) To discriminate between p_1 and p_3, we require a multi-layered cost function taking also distance into account, and (b) planner like BIT* and RRT*, which support only *additive* costs, cannot discriminate between p_1 and p_2. Our planner LA-RRT explicitly works on *non-additive* costs, and can thereby correctly identify p_1 as low cost action.

related to the additive nature of this cost function. Planners like BIT* and RRT* assume that cost terms can be summed along path segments [12]. However, such an additive cost function c_{additive} cannot discriminate between path segments which have equivalent actions along subsequent edges (p_1 versus p_2 in Fig. 4). Planners running with c_{additive} will be able to find the right *equivalence class* of solution paths (p_1 or p_2), but only by using the non-additive version of c_{actions} can we pick the correct solution path (p_1). This issue forces us to develop a dedicated planning algorithm which can correctly exploit the non-additive nature of c_{actions}.

Having defined the factored state space, we can define the problem of solving a rearrangement puzzle as a factored state space X together with a start configuration x_s, a goal region X_G, and a cost function c_{actions}. Our goal is to find a path from x_s to X_G minimizing c_{actions}.

IV. LESS-ACTIONS RRT

Less-actions RRT (LA-RRT) is a bi-directional planner modelled after RRT-Connect [18] and RRT* [12] to efficiently search over factored state spaces with non-additive action costs. LA-RRT differs by using a different extend method and a novel path optimization method, which we call *path defragmentation*.

An overview of LA-RRT is given in Alg. 2. As in RRT-Connect [18], we initialize a start tree T_a and keep a set of goal trees T_b from at most M sampled states in our goal region (Line 2). We define the variable BESTCOST as the best action cost and P_{best} as the best solution path found (Line 3–4). Like in RRT-Connect [18], we alternate tree expansion by sampling a random motion x_{rand} (Line 6) and extend the corresponding tree (Line 7). Once the trees can be connected (Line 8), we apply the path defragmentation algorithm to the solution path (see Sec. IV-B). If the new path has a better cost than the current best cost, we save P_{tmp} as the new best path (Line 11–13), and continue searching until we reach the planner termination condition PTC.

Algorithm 2: LA-RRT

Input: PTC, x_{start}, X_{goal}
1 T_a.init(x_{start}), T_b.init(X_{goal})
2 BESTCOST $\longleftarrow \infty$
3 $P_{\text{best}} \longleftarrow [\ ]$
4 **while** ¬PTC **do**
5     $x_{\text{rand}} = $ SAMPLECONFIGURATION()
6     $treeInfo \longleftarrow$ FACTOREXTEND(T_a, x_{rand})
7   **if** $treeInfo \neq TRAPPED$ **then**
8   **if** $treeInfo = REACHED$ **then**
9           $P_{\text{tmp}} \longleftarrow$ PATH(T_a, T_b)
10   PATHDEFRAGMENTATION(P_{tmp})   // IV-B
11   **if** COST(P_{tmp}) < BESTCOST **then**
12   BESTCOST = COST(P_{tmp})
13             $P_{\text{best}} = P_{\text{tmp}}$

14   SWAP(T_a,T_b)
15 **return** P_{best}

A. Factor extend and splitting edges

Contrary to RRT-Connect, LA-RRT needs to take the factors into account when extending states. This is accomplished by the FACTOREXTEND method (Alg. 3). FACTOREXTEND extends a random sample x_{rand} by doing a factor interpolation to find a new state x_{new} (Line 2), and checks if it is valid (Line 3), as in the original RRT-Connect [18].

If the edge is valid, we call the SPLITEDGE method, which splits an edge into a sequence of sub-edges, whereby each sub-edge only changes one factor at a time (Line 4-6). This is visualized in Fig. 5. By splitting the edge, we thus ensure that each sub-edge has minimal action cost. If the result is collision-free, we add the sub-edges to our tree (Line 5), or return with a failure (Line 6). If we successfully added the sub-edges to the tree, we finally return the status of the extension (Line 7-9).

B. Path defragmentation

The obtained paths using the factored extend method are often highly fragmented, meaning they exhibit frequent factor switches. To tackle this issue, we develop the PATHDEFRAG-MENTATION method. This method takes as input a path and reduces its number of factor switches.

This path defragmentation method is summarized in Alg. 4. We assume that input paths only contain edges changing at
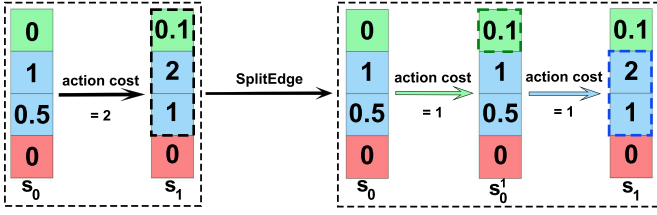
Figure 5: Example of the SPLITEDGE process ensuring minimal action cost. The edge between states s_0 and s_1 (left box) has an action cost of 2 changing the green and blue factors. Each color represents a factored state space with values corresponding to object joint values. Applying SPLITEDGE creates two edges, which changes first the green factor space, then the blue factor space. This results in two states, s_0^1 and s_1.

Algorithm 3: FACTOREXTEND

Input: T, x_{rand}
/* Factored extension of x_{rand} */
1   $x_{\text{near}} \longleftarrow T.nearest(x_{\text{rand}})$
2   $x_{\text{new}} \longleftarrow$ FACTORINTERPOLATE$(x_{\text{near}}, x_{\text{rand}})$
3   **if** $\neg$ISMOTIONVALID$(x_{\text{near}}, x_{\text{new}})$ **then return** $TRAPPED$
   /* Splitting edge into factors */
4   $X_{iso} \longleftarrow$ SPLITEDGE$(x_{\text{near}}, x_{\text{new}})$
5   **if** ARESTATESVALID$(x_{\text{near}}, X_{iso})$ **then** $T.add(X_{iso})$
6   **else return** $TRAPPED$
   /* Return status of extension */
7   **if** DISTANCE$(x_{\text{near}}, x_{\text{rand}}) > maxDistance$ **then**
8 | **return** $ADVANCED$
9   **else return** $REACHED$

---

most a single factor at a time. For simplification, we say that each edge has an associated factor, meaning the edge leads to changes in a particular factor space.

We start with an input path P containing multiple factor switches. We obtain all edges in P (Line 3) and set the first edge to e_{start} (Line 4). We then iterate over all edges until we reach the last edge (Line 5). During the iteration, we first check if the next edge has an equivalent factor, in which case we continue (Line 6-8).

If the factors of e_{start} and the next edge mismatch, we identify the next block of edges with factors identical to e_{start} (Line 9-11). First, a forward search (Line 9) finds the edge e_{end}, which is the first edge *after* the next consecutive block of the edges having the same factor as e_{start} (see Fig. Fig. 6). We then extract this block of edges (Line 10), together with the edges which differ in between (Line 11), storing them in the variables E_{same}, and E_{diff}, respectively. The resulting sets of edges are shown in Fig. 6.

Given E_{same} and E_{diff}, we try to reorder them as shown in Fig. 6. To reorder the edges, first we rewire the current edge e_{start} to the set E_{same} and then to the set E_{diff} (Line 12). The function REORDEREDGES also does collision checking during the reordering process and returns a reordered set of edges (a path segment) if all edges from E_{same} and E_{diff} have been successfully connected. If a collision occured or edges failed to get connected, we return an empty path. If the reordering is successful (Line 13), the function REPLACEPATHEDGES is called (Line 14) to replace the corresponding edges in the original path with the reordered edges. Finally, we update e_{start} by setting it to the first edge of E_{diff} (Line 15) as shown in the
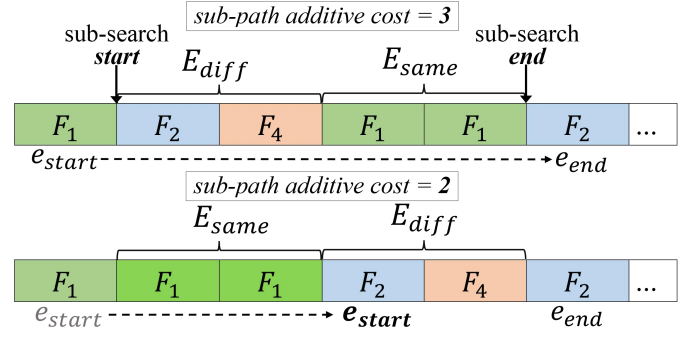

Figure 6: Example of one successful rewiring iteration in PATHDE-FRAGMENTATION method. Each block represents an edge between two states in which only a single factor is changed, whereby equivalent factor spaces have the same color. The upper path has a cost of 3 between states 1 to 5. After one iteration, the additive cost decreases to 2 in the lower path.

lower part of Fig. 6. This process is repeated until we cannot improve the cost (Line 16).

After convergence, we use two methods as post-optimization steps. First, we utilize the TRYSKIPFACTOR method in which we try to skip factors that are not mandatory to reach the goal state (Line 16). Those are removed from the path. Second, we use the SIMPLIFYACTIONINTERVALS (Line 17), where we attempt shortcuts between the same edges having the same factor switches (Fig. 6).

Algorithm 4: PATHDEFRAGMENTATION

Input: P
1   **do**
2 | $\text{cost}_{\text{old}} \longleftarrow$ COST(P)
3 | $E \longleftarrow$ GETEDGES(P)
4 | $e_{\text{start}} \longleftarrow E(0)$
5 | **while** NEXT$(e_{\text{start}}) \neq NULL$ **do**
6 | | **if** EQUIVALENTFACTORS$(e_{\text{start}}, \text{NEXT}(e_{\text{start}}))$ **then**
7 | | | $e_{\text{start}} \longleftarrow$ NEXT(e_{start})
8 | | | CONTINUE
9 | | $e_{\text{end}} \longleftarrow$ FINDENDEDGE(e_{start})
10 | | $E_{\text{same}} \longleftarrow$ SAMEFACTOREDGES$(e_{\text{start}}, e_{\text{end}})$
11 | | $E_{\text{diff}} \longleftarrow$ DIFFFACTOREDGES$(e_{\text{start}}, e_{\text{end}})$
12 | | $P' \longleftarrow$ REORDEREDGES$(E_{\text{same}}, E_{\text{diff}})$
13 | | **if** P' **then**
14 | | | REPLACEPATHEDGES$(P, P', e_{\text{start}}, e_{\text{end}})$
15 | | $e_{\text{start}} \longleftarrow$ UPDATESTARTEDGE(e_{start}, P')
16   **while** COST$(P) < \text{cost}_{\text{old}}$
17   TRYSKIPFACTOR(P)
18   SIMPLIFYACTIONINTERVALS(P)

V. MANIPULATION WITH LA-RRT

After LA-RRT converges, we use the attained factored paths to manipulate the objects with a Fetch robot as shown in Fig. 1. This requires that we plan pick and place motions of the robot for each segment of the solution path, to actually actuate the objects. The manipulation of the results consists of four main steps. We first extract the actions from the solution path and we match it with the objects in the environment. Then we compute valid random grasp positions which are simultaneously feasible at the start and goal state of the object.

(a) Maze 3 Doors

(b) Maze Slider & Obstacle

(c) Maze 4 Sliders

(d) Maze Vertical
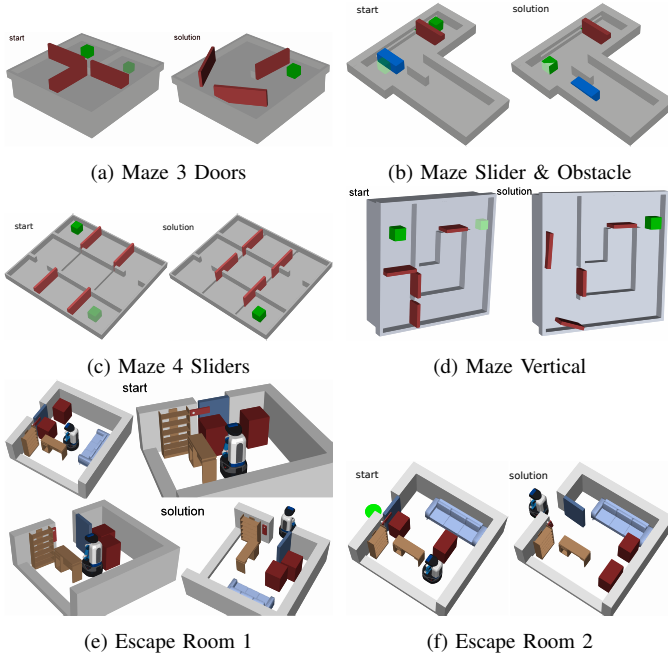
(e) Escape Room 1

(f) Escape Room 2

Figure 7: Experiments

Finally, we create a task-space region [2] which encodes the constraints to transport the object. We then plan for this motion with KPIECE [33]. If successful, we advance to the next action, or we try again by grasping a different point on the object. Note that this is just one possible way to exploit paths from LA-RRT for manipulation planning. Integrating those paths into more powerful manipulation frameworks [38], [7] could yield improved results.

VI. EVALUATION

In this section, we compare our algorithm and benchmark the success rate, cost, and solution time on the six following environments. For each environment, we define manually the factors for our factor state space.

- *Maze 3 Doors* (Fig. 7a): The green cube has to move to a goal position (transparent green). Three doors with a hinge block its way, each rotatable from $-90°$ to $90°$. The best action cost to solve this puzzle is 4.
- *Maze Slider with Obstacle* (Fig. 7b): The green cube is blocked by a sliding door that is movable only on the x-axis and a freely movable cuboid on the x-y-axis inside the walls. This puzzle demonstrates that an object may need to be moved multiple times to reach the goal. The best action cost is 3.
- *Maze 4 Sliders* (Fig. 7c): The cube has to move to the goal position through multiple sliding doors, and each door is completely movable along its rail. In the best case, each door would be moved only once so that the cube can directly be moved to the goal position, summing up to a total action cost of 5.
- *Maze Vertical* (Fig. 7d): Similar to the 3 Doors example, each door is rotatable between $[-90°, 90°]$ except the door on the top right which is rotatable between $[-90°, 0°]$. The best cost, in this case, is 3.

- *Escape Room 1-2* (Figs. 7e and 7f): In this scenario, a robot has to escape a room with fixed obstacles (desk, couch), and movable objects (cubes). The room has a lock and a door which can be rotated by $[0°, 90°]$ degrees. To account for the robot geometry, we add the robot as an additional movable object into the scene. For Escape Room 1 the best action cost is 4, consisting of moving only the cube in front of the door, unlocking the door, opening the door, and leaving the room. For Escape Room 2, it is 5 since both cubes need to be moved at least once.

A. Implementation

For the simulations, we use DART [20] available through the Robowflex [13] library. All evaluations are performed for 100 runs with a time limit of 100 seconds (except the escape room with a limit of 300 seconds) using the benchmarking tools of OMPL [34], [24]. The goal of each scenario is to find the minimal number of actions, such that the robot needs the minimal number of pick-place actions to finish its task. We benchmark LA-RRT against BIT* [5], AIT* [32], ABIT* [31], RRT* [12], and LBTRRT* [26] with default parameters using the factored state-space and the corresponding interpolate function. Because all those planners require an additive cost (Sec. III), we use the additive action cost for them. Only LA-RRT uses the non-additive action cost. For every planner, a goal space is defined for all goal indices, which leaves non-goal indices unspecified. The maximum number of states sampled in this goal space, K, is set to $K = 10$. For visualization, we color all links actuated by goal-index joints in green and all other links in red. The desired goal configurations for the goal-index joints are shown in transparent green.

Since the result of those runs need to be sent to the robot, it is crucial to find the minimal number of actions. For this purpose, we only rely on optimal planners. Indeed, while other planners might find a feasible solution faster, this solution would not be valuable, since the action cost might lead to excessive pick-place actions which we would need to execute with the robot.

B. Experimental Results

Fig. 8 presents the benchmarking results of solving the problem in the object-only space. This is obtained from each problem environment, as shown in Fig. 7. The x-axis shows the runtime in seconds, and the y-axis shows the average success rate and solution cost.

In terms of success rate, LA-RRT achieves 100% success in each environment except for *Escape Room 1* (94%) and *Escape Room 2* (98%). The RRT-based algorithms LBTRRT* and RRT* struggle to find a solution in the given time. BIT* and ABIT* achieve the best success rate in the fastest time, while AIT* performs a bit slower. In *Maze 3 Doors* and *Maze Vertical*, LA-RRT performs as fast as BIT* and ABIT*, but performs slower in the other *Maze* environments and the *Escape Room* scenarios. However, the quick success of BIT* and ABIT* comes at the price of a higher solution cost.

In every scenario, LA-RRT is able to exploit the non-additive action cost, and thereby reaches a significant lower solution cost than the next best planner. This situation is
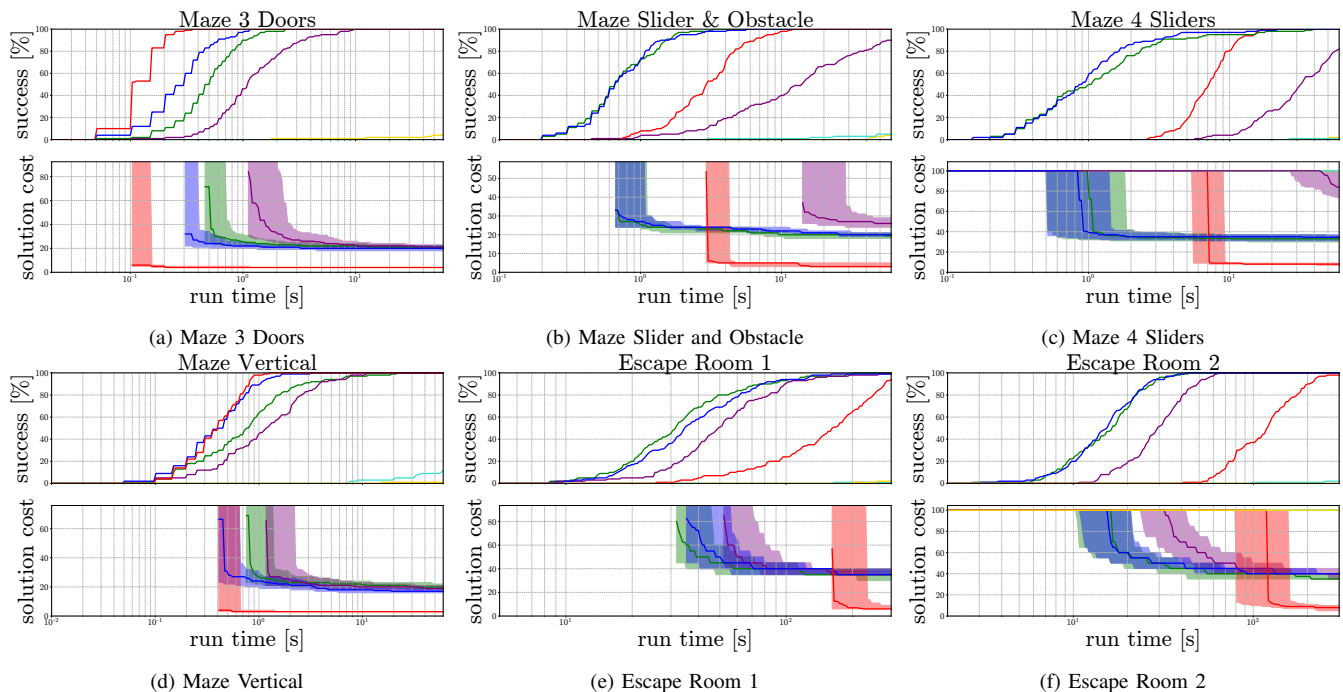
Figure 8: Experimental Results: ■ LA-RRT, ■ BIT*, ■ ABIT*, ■ LBTRRT, ■ RRT*, and ■ AIT*.

Environment	Next Best Alg.	LA-RRT	Improvement
Maze 3 Doors	19.74 (BIT*)	3.00	6.58
Maze Slider & Obstacle	19.2 (ABIT*)	3.62	5.30
Maze 4 Sliders	33.39 (ABIT*)	7.79	4.29
Maze Vertical	17.56 (BIT*)	3.02	5.81
Escape Room 1	35.40 (AIT*)	7.11	4.98
Escape Room 2	32.5 (ABIT*)	8.12	4.01

Table I: Average solution cost of LA-RRT and the next best algorithm for each environment.

summarized in Tbl. I, which shows the improvement in action cost compared to the next best planner. It can be seen that we can improve the action cost by 4.01 to 6.58 times, which is a significant improvement if we want to use those paths for manipulation tasks. As we discuss in Sec. III, other optimal planners cannot discriminate between paths with subsequent equivalent actions (Fig. 4), and therefore are not able to pick the correct equivalence class of paths, but not necessarily the true optimal solution.

While LA-RRT provides an admissible heuristic for the overall problem (i.e., a necessary condition to solve it), it can fail to produce a manipulatable solution. Fig. 9 shows two scenarios, where the Fetch robot has to reach a green goal configuration, while opening doors and removing a red cube. In Fig. 9a, the scenario is solvable by LA-RRT, but the resulting solution is not executable by the Fetch robot. Such situations could be overcome by backtracking on the manipulation level. In Fig. 9b, the red cube blocks the door and makes the scenario infeasible. This is a well-known limitation of sampling-based planners, and could be addressed by improving infeasibility checking [21]. Despite those limitations, having a good, informed admissible heuristic is important to efficiently solve difficult rearrangement puzzles.

In summary, we successfully showed that LA-RRT can be used to solve rearrangement puzzles so that final paths with low action costs can be found, making them executable by a robot in a realistic time. We believe this to be an important step towards an integrated framework for efficient planning of high-dimensional rearrangement puzzles.

VII. DISCUSSION AND CONCLUSION

We proposed to solve rearrangement puzzles using a new factored state space, which reflects the capabilities of the robot, without explicitly accounting for it. To properly exploit this state space, we developed the less-actions RRT (LA-RRT), which uses a path defragmentation method to optimize for a minimal number of actions, such that we minimize the number of pick-place actions our robot has to execute. In our evaluations, we showed that LA-RRT can consistently find lower number of actions compared to other state-of-the-art planners.

REFERENCES

[1] J. Barry, L. P. Kaelbling, and T. Lozano-Pérez, "A hierarchical approach to manipulation with diverse actions," in *IEEE International Conference on Robotics and Automation*, 2013, pp. 1799–1806.

[2] D. Berenson, S. Srinivasa, and J. Kuffner, "Task space regions: A framework for pose-constrained manipulation planning," *International Journal of Robotics Research*, vol. 30, no. 12, pp. 1435–1460, 2011.

[3] J. Cortés, L. Jaillet, and T. Siméon, "Molecular disassembly with RRT-like algorithms," in *IEEE International Conference on Robotics and Automation*. IEEE, 2007, pp. 3301–3306.

[4] D. Driess, J.-S. Ha, and M. Toussaint, "Deep visual reasoning: Learning to predict action sequences for task and motion planning from an initial scene image," in *Robotics: Science and Systems*, 2020.
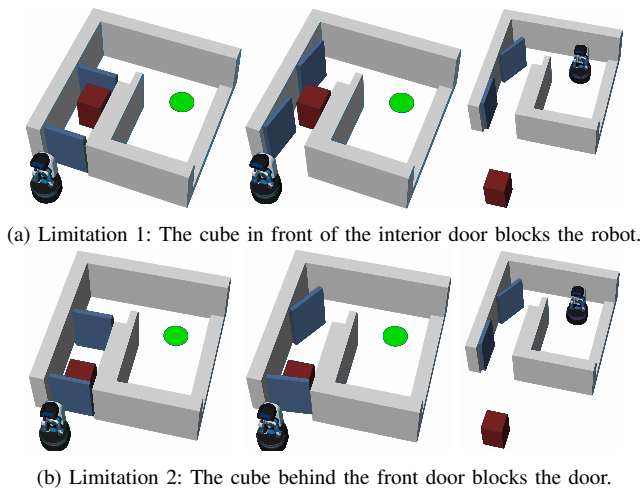
(a) Limitation 1: The cube in front of the interior door blocks the robot.

(b) Limitation 2: The cube behind the front door blocks the door.

Figure 9: Environments solvable by LA-RRT but are not feasible.

[5] J. D. Gammell, T. D. Barfoot, and S. S. Srinivasa, "Batch informed trees (BIT*): Informed asymptotically optimal anytime search," *The International Journal of Robotics Research*, vol. 39, no. 5, pp. 543–567, 2020.

[6] 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.

[7] C. R. Garrett, T. Lozano-Pérez, and L. P. Kaelbling, "FFRob: Leveraging symbolic planning for efficient task and motion planning," *International Journal of Robotics Research*, vol. 37, no. 1, pp. 104–136, 2018.

[8] ——, "Sampling-based methods for factored task and motion planning," *The International Journal of Robotics Research*, vol. 37, no. 13-14, pp. 1796–1825, 2018.

[9] F. Grothe, V. N. Hartmann, A. Orthey, and M. Toussaint, "ST-RRT*: Asymptotically-optimal bidirectional motion planning through space-time," in *IEEE International Conference on Robotics and Automation*. IEEE, 2022.

[10] V. N. Hartmann, A. Orthey, D. Driess, O. S. Oguz, and M. Toussaint, "Long-horizon multi-robot rearrangement planning for construction," *IEEE Transactions on Robotics*, pp. 1–14, 2022.

[11] L. P. Kaelbling and T. Lozano-Pérez, "Hierarchical planning in the now," in *Workshops at the Twenty-Fourth AAAI Conference on Artificial Intelligence*, 2010.

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

[13] Z. Kingston and L. E. Kavraki, "Robowflex: Robot motion planning with MoveIt made easy," in *IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2022.

[14] ——, "Scaling multimodal planning: Using experience and informing discrete search," *IEEE Transactions on Robotics*, pp. 1–19, 2022.

[15] Z. Kingston, M. Moll, and L. E. Kavraki, "Sampling-based methods for motion planning with constraints," *Annual review of control, robotics, and autonomous systems*, vol. 1, pp. 159–185, 2018.

[16] Y. Koga and J.-C. Latombe, "On multi-arm manipulation planning," in *IEEE International Conference on Robotics and Automation*. IEEE, 1994, pp. 945–952.

[17] A. Krontiris and K. E. Bekris, "Dealing with difficult instances of object rearrangement." in *Robotics: Science and Systems*, vol. 1123, 2015.

[18] J. J. Kuffner and S. M. LaValle, "RRT-connect: An efficient approach to single-query path planning," in *IEEE International Conference on Robotics and Automation*, vol. 2. IEEE, 2000, pp. 995–1001.

[19] S. M. LaValle, *Planning Algorithms*. Cambridge University Press, 2006.

[20] J. Lee, M. X. Grey, S. Ha, T. Kunz, S. Jain, Y. Ye, S. S. Srinivasa, M. Stilman, and C. K. Liu, "Dart: Dynamic animation and robotics toolkit," *Journal of Open Source Software*, vol. 3, no. 22, p. 500, 2018.

[21] S. Li and N. Dantam, "Learning proofs of motion planning infeasibility." in *Robotics: Science and Systems*, 2021.

[22] J. Mirabel and F. Lamiraux, "Manipulation planning: addressing the crossed foliation issue," in *IEEE International Conference on Robotics and Automation*. IEEE, 2017, pp. 4032–4037.

[23] ——, "Handling implicit and explicit constraints in manipulation planning," in *Robotics: Science and Systems*, 2018, p. 9p.

[24] M. Moll, I. A. Şucan, and L. E. Kavraki, "Benchmarking motion planning algorithms: An extensible infrastructure for analysis and visualization," *IEEE Robotics & Automation Magazine*, vol. 22, no. 3, pp. 96–102, September 2015.

[25] A. Orthey, S. Akbar, and M. Toussaint, "Multilevel motion planning: A fiber bundle formulation," 2020, arXiv:2007.09435 [cs.RO].

[26] O. Salzman and D. Halperin, "Asymptotically near-optimal RRT for fast, high-quality motion planning," *IEEE Transactions on Robotics*, vol. 32, no. 3, pp. 473–483, 2016.

[27] P. S. Schmitt, W. Neubauer, W. Feiten, K. M. Wurm, G. V. Wichert, and W. Burgard, "Optimal, sampling-based manipulation planning," in *IEEE International Conference on Robotics and Automation*. IEEE, 2017, pp. 3426–3432.

[28] R. Shome, D. Nakhimovich, and K. E. Bekris, "Pushing the boundaries of asymptotic optimality in integrated task and motion planning," in *Workshop on the Algorithmic Foundations of Robotics*. Springer, 2020, pp. 467–484.

[29] T. Siméon, J.-P. Laumond, J. Cortés, and A. Sahbani, "Manipulation planning with probabilistic roadmaps," *International Journal of Robotics Research*, vol. 23, no. 7-8, pp. 729–746, 2004.

[30] M. Stilman and J. Kuffner, "Planning among movable obstacles with artificial constraints," *The International Journal of Robotics Research*, vol. 27, no. 11-12, pp. 1295–1307, 2008.

[31] M. P. Strub and J. D. Gammell, "Advanced BIT* (ABIT*): Sampling-based planning with advanced graph-search techniques," in *IEEE International Conference on Robotics and Automation*. IEEE, 2020, pp. 130–136.

[32] ——, "Adaptively informed trees (AIT*) and effort informed trees (EIT*): Asymmetric bidirectional sampling-based path planning," *The International Journal of Robotics Research*, vol. 41, no. 4, pp. 390–417, 2022.

[33] I. A. Şucan and L. E. Kavraki, "Kinodynamic motion planning by interior-exterior cell exploration," in *Algorithmic Foundation of Robotics VIII*. Springer, 2009, pp. 449–464.

[34] 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, https://ompl.kavrakilab.org.

[35] W. Thomason and R. A. Knepper, "A unified sampling-based approach to integrated task and motion planning," in *International Symposium of Robotics Research*. Springer, 2019, pp. 773–788.

[36] W. Thomason, M. P. Strub, and J. D. Gammell, "Task and motion informed trees (TMIT*): Almost-surely asymptotically optimal integrated task and motion planning," *IEEE Robotics and Automation Letters*, vol. 7, no. 4, pp. 11 370–11 377, 2022.

[37] M. Toussaint, "Logic-geometric programming: An optimization-based approach to combined task and motion planning," in *International Joint Conference on Artificial Intelligence*, 2015.

[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 *Robotics: Science and Systems*, 2018.

[39] W. Vega-Brown and N. Roy, "Asymptotically optimal planning under piecewise-analytic constraints," in *Algorithmic Foundations of Robotics XII*. Springer, 2020, pp. 528–543.

[40] R. Wang, K. Gao, D. Nakhimovich, J. Yu, and K. E. Bekris, "Uniform object rearrangement: From complete monotone primitives to efficient non-monotone informed search," in *IEEE International Conference on Robotics and Automation*. IEEE, 2021, pp. 6621–6627.

[41] R. Wang, Y. Miao, and K. E. Bekris, "Efficient and high-quality prehensile rearrangement in cluttered and confined spaces," in *IEEE International Conference on Robotics and Automation*. IEEE, 2022, pp. 1968–1975.