Uniform Object Rearrangement: From Complete Monotone Primitives to Efficient Non-Monotone Informed Search

Rui Wang*, Kai Gao*, Daniel Nakhimovich*, Jingjin Yu and Kostas E. Bekris

***Abstract*— Object rearrangement is a widely-applicable and challenging task for robots. Geometric constraints must be carefully examined to avoid collisions and combinatorial issues arise as the number of objects increases. This work studies the algorithmic structure of rearranging uniform objects, where robot-object collisions do not occur but object-object collisions have to be avoided. The objective is minimizing the number of object transfers under the assumption that the robot can manipulate one object at a time. An efficiently computable decomposition of the configuration space is used to create a "region graph", which classifies all continuous paths of equivalent collision possibilities. Based on this compact but rich representation, a complete dynamic programming primitive $\texttt{DFS}_{\texttt{DP}}$ performs a recursive depth first search to solve monotone problems quickly, i.e., those instances that do not require objects to be moved first to an intermediate buffer. $\texttt{DFS}_{\texttt{DP}}$ is extended to solve single-buffer, non-monotone instances, given a choice of an object and a buffer. This work utilizes these primitives as local planners in an informed search framework for more general, non-monotone instances. The search utilizes partial solutions from the primitives to identify the most promising choice of objects and buffers. Experiments demonstrate that the proposed solution returns near-optimal paths with higher success rate, even for challenging non-monotone instances, than other leading alternatives.**

I. Introduction

Object rearrangement is a critical robot skill broadly applicable in the logistics, industrial, and service domains. For instance, robots can rearrange merchandise in grocery shelves as in Fig. 1(left), retrieve food in packed fridges for home automation, or perform packaging of products for shipping [1]. This work focuses on problems where one object is manipulated at a time without incurring any object-object collisions. The objective is to minimize the number of object transfers needed to complete a rearrangement. The setting is akin to an attendant moving cars in a crammed parking lot. Clearly, collisions between two cars should not occur and the attendant should minimize the number of times he drives a car. Similar scenarios occur in manipulation, e.g., when a soda must be retrieved from a fridge, multiple other beverages may have to be rearranged first. While the arm may be able to reach the objects with an overhand grasp as in Fig. 1(right), it may not be possible to just lift objects to avoid collisions among them. Furthermore, given the sometimes unpredictable effects of object-object collisions, these collisions should be avoided.

*The first three authors contributed equally to this paper. The authors are with the Department of Computer Science, Rutgers University, NJ, USA. Email: {rw485, kg627, dn332, jy512, kb572}@cs.rutgers.edu. The work is supported in part by NSF awards IIS-1845888, CCF-1934924 and an NSF NRT project 2021628.

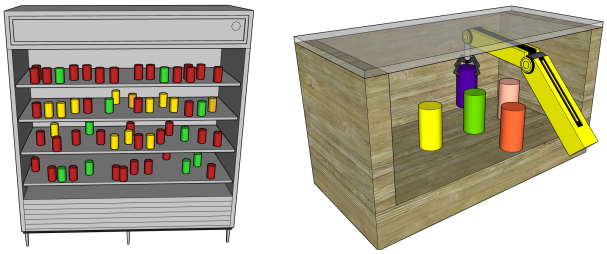

Fig. 1. (left) Robots in logistics have to rearrange similar objects in shelves. (right) The focus is on combinatorial and geometric aspects of rearrangement when the arm can reach objects without colliding with them but cannot lift objects to guarantee they do not collide with each other.

A rearrangement domain, where performance guarantees can be argued but which is already hard, involves tabletop setups and overhand grasps, where both robot-object and object-object collisions can be ignored [2]. This work pushes forward the understanding of the algorithmic structure of rearrangement by considering object-object collisions for uniform-shaped objects in planar setups. A principled solution pipeline is proposed (Fig. 2), which yields complete and efficient primitives for monotone instances and an effective informed search framework that quickly computes high-quality non-monotone solutions. In a monotone instance, each object needs to be moved at most once, i.e., without having to move first to an intermediate, buffer location. The pipeline is composed of three key components/contributions:

1. A decomposition of the space into equivalent regions resulting in a compact *region graph* containing all possible object paths in terms of collision sets that can arise. It abstracts away the problem's continuous, geometric aspects.

2. A novel *dynamic programming* routine $\texttt{DFS}_{\texttt{DP}}$ that solves monotone instances quickly by reasoning over object paths on the region graph. It's extended to optimally solve 1-buffer, non-monotone problems for an object and buffer choice.

3. An *informed search* framework that uses $\texttt{DFS}_{\texttt{DP}}$ variations as local planners. Given partial solutions generated by $\texttt{DFS}_{\texttt{DP}}$, it generates heuristics to explore promising objects and buffers for quickly making progress towards the goal arrangement.

Simulation experiments show the efficiency of the monotone solver $\texttt{DFS}_{\texttt{DP}}$. The informed search is shown to be faster and to return higher-quality solutions than alternatives for non-monotone instances. For problems with 1 or 2 buffers where the optimal solution can be discovered, the framework returns almost optimal solutions. An ablation study compares against a baseline search that uses the monotone solver as a local planner. It highlights the benefits of the non-monotone extension of $\texttt{DFS}_{\texttt{DP}}$ and of the chosen heuristics.

II. Related Work

An apparently simple setup involves picking objects from tabletops, which are lifted sufficiently high before being placed back to avoid object-object collisions [2]–[5]. This setting is as computationally hard to solve optimally as the Travelling Salesperson Problem (TSP) when objects' starts and goals do not overlap [2]. Overlapping starts and goals complicates the problem as it reduces to the Feedback Vertex Set (FVS) problem [6], which is possibly APX-hard. Dual arm rearrangement allows for parallelism but complicates reasoning [3]. Integer programming is often applied for deciding the object order together with motion planning [2], [3]. Optimal tabletop placement has also been approached via Answer Set Programming (ASP) [4], [5] and informed heuristics [7]. Similarly to this work, these efforts aim to minimize object grasps/pushes but address packing new items into cluttered environments; this is more akin to the unlabeled version of this works' problem.

For more confined spaces, small monotone problems have been addressed via backtracking search [8]. A useful structure for solving problems in the general case is a "dependency graph" [9], which expresses constraints between objects given their starts and goals. Acyclic graphs indicate existence of monotone solutions. The "true" dependency graph is difficult to construct in general as all object paths must be considered. With approximations for Minimum Constraint Removal (MCR) paths [10], it is possible to practically build good dependency graphs [11], [12] to solve general instances. This paper focuses on uniform disc-shaped object instances and object-object interactions. By constructing a "region graph" that compactly represents all object paths, it is possible to search over dependency graphs with efficient and complete solvers for monotone and non-monotone instances.

Solutions for integrated task and motion planning (TAMP) can be applied to general rearrangement [13], [14]. While they incorporate heuristics and are probabilistically complete (PC), it is difficult to make arguments about optimality. Furthermore, insights from rearrangement planners can lead to effective heuristics for TAMP. Pushing allows simultaneous action on multiple objects [15]–[17] though actions may be irreversible. Pushing has been studied in the context of robust rearrangement under uncertainty [18]–[20].

Navigation Among Movable Obstacles (NAMO) [21] is related to rearrangement. It is NP-hard [22] and its difficulty depends on linearity and monotonicity notions. A problem is linear if collision free components can be traversed in sequence, where earlier actions do not constrain future ones. For non-monotone, non-linear NAMO problems, a PC algorithm exists for axis-aligned objects and robots [23] though it may return highly-redundant paths. Recently, methods have tackled online NAMO settings [24]–[26]. In object retrieval, movable obstacles may obstruct paths and works have used dependency reasoning to generate valid plans [18], [27] and scale linearly in actions with the number of objects [28]. Recently, algorithms have been proposed to explicitly minimize the number of objects to relocate [29], [30].

III. Problem Setup and Notation

Let $\mathcal{W} \subset \mathbb{R}^2$ be a bounded polygonal region where n labeled uniform-shaped objects $\mathcal{O} = \{o_1, \cdots, o_n\}$ reside. An arrangement $\mathcal{A}$ of $\mathcal{O}$ is given as $(p_1, \cdots, p_n) \in \mathcal{W}^n$, where $p_i \in \mathcal{W}$ defines the position of o_i, i.e., the coordinates of o_i's center. $\mathcal{A}[o_i] = p_i$ indicates that object o_i assumes position p_i in the arrangement $\mathcal{A}$. Define as $V(p)$ the subset of $\mathcal{W}$ occupied by an object at position p. An arrangement $\mathcal{A}$ is *feasible* if no object-object collisions occurs, i.e., $\mathcal{A}$ is feasible if $\forall i, j \in [1, n], i \neq j : V(\mathcal{A}[o_i]) \cap V(\mathcal{A}[o_j]) = \emptyset$.

A robotic arm can reach objects at any position $p \in \mathcal{W}$ without colliding with them. Given an arrangement $\mathcal{A}$, the arm can move one object at a time from its current position $p_i = \mathcal{A}[o_i]$ to a new position p_i', giving rise to a new arrangement $\mathcal{A}'$, where $\mathcal{A}'[o_i] = p_i'$ and $\forall j \in [1, n], j \neq i : \mathcal{A}'[o_j] = \mathcal{A}[o_j]$. The arm's motion results in continuous paths $\pi_i : [0, 1] \to \mathcal{W}$ for object o_i with $\pi_i(0) = p_i$ and $\pi_i(1) = p_i'$. The arm cannot raise the picked object far enough to guarantee collision avoidance with other objects. Consequently, object paths can be split into valid and non-valid paths. A path π for moving object o_i is *valid* if it does not result in a collision between o_i and all the other objects o_j given their positions in a feasible $\mathcal{A}$, where $1 \leq j \leq n, j \neq i$.

A candidate new position $p \in \mathcal{W}$ for object o_i given arrangement $\mathcal{A}$, may cause object o_i to collide with other objects. The set of objects that collide with o_i given its position p is called as an *interference set* for p. Specifically, the interference set for position p of object o_i given arrangement $\mathcal{A}$ is defined as $\mathbf{I}_{i,\mathcal{A}}(p) = \{o_j \in \mathcal{O}, o_j \neq o_i : V(p) \cap V(\mathcal{A}[o_j]) \neq \emptyset\}$. This notion can be extended to a set of arrangements $\mathbf{A}$ as $\mathbf{I}_{i,\mathbf{A}}(p) = \bigcup_{\mathcal{A} \in \mathbf{A}} \mathbf{I}_{i,\mathcal{A}}(p)$. Furthermore, given a path π_i for object o_i, the union of interference sets along the path π_i contains all objects that o_i will collide with along π_i. This is defined as the interference set $\mathbf{I}_{i,\mathcal{A}}(\pi_i)$ of path π_i, i.e., $\mathbf{I}_{i,\mathcal{A}}(\pi_i) = \bigcup_{\forall p \in \pi_i} \mathbf{I}_{i,\mathcal{A}}(p)$.

The rearrangement problem considered here is to discover a sequence of valid object paths which bring objects $\mathcal{O}$ from an initial feasible arrangement $\mathcal{A}_I$ to a final feasible arrangement $\mathcal{A}_F$. The concatenation of such valid paths gives rise to a solution path sequence Π. Optimal solution path sequences Π^* minimize the number of object paths needed to solve a rearrangement problem.

This definition transforms the rearrangement problem into a path planning problem on the arrangement space. An *arrangement space* is the space of all feasible object arrangements. For arbitrary arrangements $\mathcal{A}_a$, $\mathcal{A}_b$, there is a transition from $\mathcal{A}_a$ to $\mathcal{A}_b$ if $\exists! o_i \in \mathcal{O}$ s.t. $\mathcal{A}_a[o_i] \neq \mathcal{A}_b[o_i]$, and there is a valid path π_i for o_i from $\mathcal{A}_a[o_i]$ to $\mathcal{A}_b[o_i]$.

An instance is *monotone* if there is a solution path sequence from $\mathcal{A}_I$ to $\mathcal{A}_F$ that contains at most one valid path π_i for each o_i. *Non-monotone* instances require solution path sequences where at least one object is moved at least twice, i.e., where the object is first placed to an intermediate *buffer* position before moved to its target.

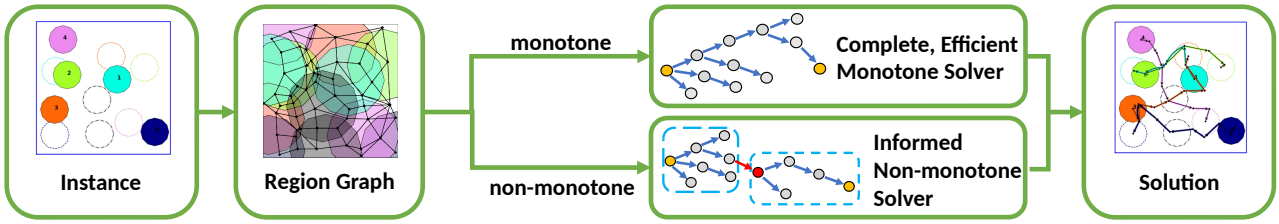

Fig. 2. A rearrangement instance is defined by the objects' starts and goals. The approach pre-allocates candidate buffers and decomposes the space to generate a region graph. Each region corresponds to a different interference set with existing start, goal and buffer locations. If a monotone solution exists given all object paths in the region graph, a dynamic programming approach discovers it efficiently. For non-monotone problems, a search method incrementally builds a tree, where nodes are arrangements and edges represent object transfers to their goals together an object moved to a buffer (called a "perturbation"). Informed heuristics based on the dynamic programming solver guide the choice of objects and buffers for faster, high-quality solutions.

IV. Region Graph

Given a set of arrangements $\mathbf{A}$ (e.g., $\{\mathcal{A}_I, \mathcal{A}_F\}$), the workspace $\mathcal{W}$ can be decomposed into a set of closed, path-connected regions $\mathcal{R}_\mathbf{A}$. Each region contains positions with the same interference set. Denote as $\mathcal{D}(p) := \{p' : V[p] \cap V[p'] \neq \emptyset\}$ the subset of $\mathcal{W}$ for which an object placement will collide with an object located at p. Alg. 1 shows the region decomposition process based on $\mathbf{A}$. For each position p in each arrangement in $\mathbf{A}$, each existing region $r \in R_\mathbf{A}$ is split by $r \cap \mathcal{D}(p)$ and $r \backslash \mathcal{D}(p)$ (lines 5-10). After that, regions that are not path-connected are split into their path-connected components (lines 11-13).

Algorithm 1: Region Decomposition($\mathbf{A}$)

```
1  M ← ∅, R_A ← ∅
2  for A ∈ A do
3      for p ∈ A do
4          for r ∈ R_A do
5              if r ∩ D(p) ≠ ∅ then
6                  R_A ← R_A ∪ {r ∩ D(p)}, r ← r \ D(p)
7                  if r is ∅ then R_A ← R_A \ {r}
8          d ← (W ∩ D(p)) \ M
9          if d ≠ ∅ then R_A ← R_A ∪ {d}
10         M ← M ∪ D(p)
11 for r ∈ R_A do
12     R_A ← R_A \ {r}
13     for c ∈ components(r) do R_A ← R_A ∪ {c}
14 return R_A
```

The *region graph* $G_\mathbf{A} = (\mathcal{R}_\mathbf{A}, E_\mathbf{A})$ has regions as nodes. An edge $(r_1, r_2) \in E_\mathbf{A}$ exists if and only if $r_1 \cup r_2$ is a path-connected subset of $\mathcal{W}$. Given a walk W on $G_\mathbf{A}$ with vertex sequence $R = (r_0, ..., r_k)$, its interference set is $\mathbf{I}(W) = \bigcup_{r_i \in R_\mathbf{A}} \mathbf{I}(r_i)$. Any continuous path in the workspace can be associated with a walk on the region graph with the same interference set. Since multiple paths in $\mathcal{W}$ may correspond to the same path in $G_\mathbf{A}$, the paths in the region graph form equivalence classes of workspace paths sharing the same interference set. Although multiple paths in the region graph can have the same interference set, a region graph for one or more arrangements categorizes all paths for an object into a finite number of classes. Thus, for any object, it is possible to systematically explore all path options between two positions in terms of interference sets.

Since the focus is on finding solutions with minimum interference sets, it is not necessary to discover every homotopy class of workspace paths traversing the regions but rather only the classes of paths with varying interference sets. Thus, in order to reduce the number of region adjacencies, it is reasonable to only check adjacency between regions pairs with interference sets differing by one object.

V. High Performance Monotone Solver

For a monotone problem, there are 2^n different arrangements, where an object can be placed either at its start or goal. Given an arrangement $\mathcal{A}$, let $\mathcal{O}(\mathcal{A}) = \{o_i : \mathcal{A}[o_i] = \mathcal{A}_F[o_i]\}$ denote the subset of objects at their goals. $\mathcal{A}$ is *accessible* if the subproblem of moving $\mathcal{O}(\mathcal{A})$ from $\mathcal{A}_I$ to $\mathcal{A}$ is monotone. If $\mathcal{A}$ is known to be accessible, solving the subproblem of moving $\mathcal{O} \backslash \mathcal{O}(\mathcal{A})$ from $\mathcal{A}$ to $\mathcal{A}_F$ is sufficient for solving the full problem. In addition, the subproblem's solution does not depend on $\mathcal{O}(\mathcal{A})$'s ordering.

Algorithm 2: $\texttt{DFS}_{\texttt{DP}}(T, \mathcal{A}_C, \mathcal{A}_F, G_{\{\mathcal{A}_I, \mathcal{A}_F\}})$

```
1  for o ∈ O \ O(A_C) do
2      A_new[O \ {o}] = A_C[O \ {o}]
3      A_new[o] = A_F[o]
4      if A_new not in T then
5          π, D_π ← RG-DFS(G_{A_I A_F}, A_C, A_new, D_π)
6          if π ≠ ∅ then
7              T[A_new].parent ← A_C
8              if A_new ≠ A_F then
9                  T, D_π = DFS_DP(T, D_π, A_new, A_F, G_{A_I, A_F})
10             if A_F ∈ T then return T, D_π
11 return T, D_π
```

Given this observation, a dynamic program $\texttt{DFS}_{\texttt{DP}}$ is presented in Alg. 2 to solve the monotone problem. It grows a search tree T in the arrangement space rooted at $\mathcal{A}_I$. Each node on the tree is accessible from the root and $\texttt{DFS}_{\texttt{DP}}$ tests its connection to $\mathcal{A}_F$ in a depth-first manner. Alg. 2 first enumerates all possible objects and attempts to move each of them from their current positions to their goals (lines 1-3). When the newly constructed arrangement $\mathcal{A}_{new}$ is not in T (line 4), a valid path for the object o from $\mathcal{A}_C[o]$ to $\mathcal{A}_F[o]$ is searched (line 5). If a valid path is found (line 6), $\mathcal{A}_{new}$ is labeled accessible and added to T (line 7). If $\mathcal{A}_{new}$ is not $\mathcal{A}_F$, the program recurses on it (lines 8-9). Otherwise, the solution is found (line 10) with corresponding object paths.

$\texttt{DFS}_{\texttt{DP}}$ searches for an object path local to $\mathcal{A}_C$ within a subgraph of the region graph $G_{\{\mathcal{A}_I, \mathcal{A}_F\}} = (\mathcal{R}_{\{\mathcal{A}_I, \mathcal{A}_F\}}, E_{\{\mathcal{A}_I, \mathcal{A}_F\}})$ constructed from $\mathcal{A}_I$ and $\mathcal{A}_F$. This subgraph ignores dependencies from non-occupied positions. Since there are 2^n different region subgraphs (one per arrangement $\mathcal{A}_C$), but only n pairs of start/goal positions, paths are stored in a dictionary D_π for each start/goal pair for future queries. Since the dictionary D_π is enriched

incrementally, time is saved in subsequent calls to $\texttt{DFS}_{\texttt{DP}}$ by first checking D_π. RG-DFS, the path finding algorithm for objects on the region graph, is only executed when no valid object path local to $\mathcal{A}_C$ was previously found.

Proposition 1. $\texttt{DFS}_{\texttt{DP}}$ *over the region graph* $G_{\{\mathcal{A}_I,\mathcal{A}_F\}} = (\mathcal{R}_{\{\mathcal{A}_I,\mathcal{A}_F\}}, E_{\{\mathcal{A}_I,\mathcal{A}_F\}})$ *is complete for monotone instances.*

Since there exists a one-to-many mapping from region graph paths to all possible object paths in $\mathcal{W}$, the path finding algorithm RG-DFS effectively searches for object paths when searching the region graph in a depth-first manner. Furthermore, the search tree is finite since the region graph is finite given a finite number of objects. Therefore, RG-DFS is complete, which leads to the completeness of $\texttt{DFS}_{\texttt{DP}}$.

In order to solve non-monotone instances, at least one object has to be moved to a buffer. This work refers to an action of moving object o_i to a buffer b_i as a *perturbation* and denoted as $P(o_i, b_i)$. A perturbation at $\mathcal{A}$ splits the movement of object o_i into two phases: $\mathcal{A}[o_i] \rightarrow b_i$ given the perturbation and $b_i \rightarrow \mathcal{A}_F(o_i)$. While a monotone solution can be treated as a permutation of n actions $a_{o_1}, ..., a_{o_n}$ where each action a_{o_i} moves o_i from $\mathcal{A}_I[o_i]$ to $\mathcal{A}_F[o_i]$, a non-monotone solution with the perturbation $P(o_i, b_i)$ can be viewed as a permutation of $n+1$ actions with partial order enforced between the two actions involving o_i. Thus, the monotone planner $\texttt{DFS}_{\texttt{DP}}$ is extended into a planner $\texttt{EDFS}_{\texttt{DP}}$, which solves 1-buffer, non-monotone problems given a specific choice of object o_i and buffer b_i to which o_i moves.

VI. Non-monotone Framework

For general non-monotone problems, solution quality and computation time are largely determined by the choice of which objects to move to a buffer and which buffer to use. Alg. 3 describes a non-monotone search for this purpose, which searches through the space of perturbations $P(o_i, b_i)$.

Algorithm 3: Informed-Search($\mathcal{A}_I$, $\mathcal{A}_F$)

```
1  T ← ∅
2  T_new = DFS_DP-LocalPlanner(A_I, A_F)
3  if A_F ∈ T_new.V then return T_new
4  T ← T + T_new
5  while T is not connected to A_F do
6      A_C ← SELECT-EXPANSION-NODE(T)
7      if A_C = ∅ then A_C ← RANDOMNODE(T.V)
8      O_C ← SELECT-PERTURBATION-OBJECT(A_C)
9      B_C ← SELECT-PERTURBATION-BUFFER(A_C, O_C)
10     for P(o_i, b_i) ∈ (O_C, B_C) do
11         T_new = EDFS_DP-LocalPlanner(A_C, A_F, P(o_i, b_i))
12         if A_F ∈ T_new.V then return T + T_new
13         else T ← T + T_new
```

The framework receives as input the problem instance $(\mathcal{A}_I, \mathcal{A}_F)$ and returns a search tree in the arrangement space with a path from $\mathcal{A}_I$ to $\mathcal{A}_F$. It initializes the search tree T (line 1) and calls the proposed monotone solver, which recursively calls Alg. 2, to return a subtree T_{new} rooted at $\mathcal{A}_I$ (line 2) and identify if the problem is monotone (line 3). If the problem is not monotone, the partial solution (subtree) T_{new} is added to the search tree (line 4) and new perturbations are attempted until a solution is found (line 5). For each perturbation, the algorithm decides on a new expansion node (the arrangement $\mathcal{A}_C$ to launch the non-monotone local planner) (line 6) and which objects $\mathcal{O}_\mathcal{C}$ (line 8) to place in which buffers $\mathcal{B}_\mathcal{C}$ (line 9). For each computed perturbation $P(o_i, b_i)$ (line 10), the non-monotone planner (e.g. $\texttt{EDFS}_{\texttt{DP}}$) attempts to solve the subproblem $\mathcal{A}_C \rightarrow \mathcal{A}_F$ with $P(o_i, b_i)$ (line 11) and a solution is returned if solved (line 12). Otherwise, a partial solution T_{new} is added to the search tree (line 13). If no expansion node is recommended, a random node in the tree will be selected for potential perturbations (line 7) in order to ensure exhaustiveness. For the accompanying experiments, the search stops when a time threshold is exceeded.

As indicated in Alg. 3, the key component of selecting a promising perturbation involves, selecting an arrangement node $\mathcal{A}_C$ to expand, deciding which objects to move to buffers, and selecting buffers to use.

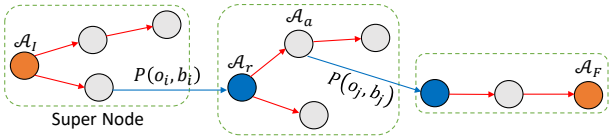

Fig. 3. A search tree built according to Alg. 3. Nodes correspond to arrangements, while directed edges correspond to monotone transitions from one arrangement to another. A super node is a set of nodes which are accessible from the same root; they are connected via a "perturbation" (blue arrow), where an object is moved to a buffer. Except for the initial arrangement $\mathcal{A}_I$, roots of super nodes (blue circles) are perturbation nodes.

A. Selecting Arrangements for Expansion

As shown in Fig. 3, the search tree can be divided into multiple subtrees, referred to as *super nodes*, based on perturbations. Each super-node is rooted at $\mathcal{A}_I$ or a *perturbation node* (blue circles), which is defined as the arrangement derived from a perturbation. Given a super node rooted at $\mathcal{A}$, other members of the super node are the nodes on the search tree which are reachable from $\mathcal{A}$ via monotone transitions, i.e., without perturbations. In Fig. 3, assuming a path from $\mathcal{A}_a$ to $\mathcal{A}_F$ can be found via the local planner $\texttt{EDFS}_{\texttt{DP}}$ given a perturbation $P(o_j, b_j)$, then that path can also be found by calling $\texttt{EDFS}_{\texttt{DP}}$ from $\mathcal{A}_r$, the root of the super node that $\mathcal{A}_a$ belongs in. If the solution path exists along some of the siblings of $\mathcal{A}_a$, but is not discoverable from $\mathcal{A}_a$, calling $\texttt{EDFS}_{\texttt{DP}}$ at $\mathcal{A}_r$ will still work. Therefore, for each super node, launching the planner from the root is the preferred way to generate solutions. Among the super-nodes, those with fewer perturbations relative to $\mathcal{A}_I$ are prioritized. Among those with the same number of perturbations, those with more objects at their goal positions are prioritized as they are more similar to $\mathcal{A}_F$.

B. Selecting Objects for Perturbation

Once an arrangement is selected to be expanded, a decision needs to be made about which objects to move to a buffer. Intuitively, a highly *constraining* object, whose current position $\mathcal{A}_C[o]$ blocks other objects to make progress to their goals, should be prioritized. Symmetrically, a highly *constrained* object, whose goal position $\mathcal{A}_F[o]$ is blocked by

other objects not at their goals, should not be considered to move directly towards its goal. In summary, the priority for selecting perturbations should be given to objects that are both highly constraining and constrained.

The extent to which an object is constraining or constrained can be measured via its degree in an approximate dependency graph, which uses the interference set $\mathbf{I}_{i,\mathcal{A}}(p)$ of the current positions of objects not yet moved to their goals in $\mathcal{A}_C$. In particular, denote $\overline{\mathcal{O}}(\mathcal{A}_C)$ as the sets of objects not yet moved to their goals at $\mathcal{A}_C$. If the current position $\mathcal{A}_C[o_i]$ of an object $o_i \in \overline{\mathcal{O}}(\mathcal{A}_C)$ interferes with the goal position $\mathcal{A}_F[o_j]$ of another object $o_j \in \overline{\mathcal{O}}(\mathcal{A}_C)$, then it creates a dependency edge $e(j,i)$, indicating that o_j depends on object o_i. An indicator variable $\mathbb{1}(j,i)$ is defined as

$$\mathbb{1}(j,i) = \begin{cases} 1, & \text{if } V(\mathcal{A}_C[o_i]) \cap V(\mathcal{A}_F[o_j]) \neq \emptyset \\ 0, & \text{otherwise.} \end{cases}$$

where $\mathbb{1}(j,i) = 1$ indicates the existence of edge $e(j,i)$.

Given all such dependency edges, define $\mathcal{D}_I(o_i) = \sum_{o_j \in \overline{\mathcal{O}}(\mathcal{A}_C)} \mathbb{1}(j,i)$ as the *inner degree of dependency* which indicates the degree of which the object o_i is constraining at $\mathcal{A}_C$ and $\mathcal{D}_O(o_i) = \sum_{o_j \in \overline{\mathcal{O}}(\mathcal{A}_C)} \mathbb{1}(i,j)$ as the *outer degree of dependency* which indicates the degree of which the object o_i is constrained at $\mathcal{A}_C$. Then, all objects in $\overline{\mathcal{O}}(\mathcal{A}_C)$ are ranked in descending order of $\mathcal{D}_I(o_i) + \mathcal{D}_O(o_i)$, so that the most constraining and constrained objects are prioritized.

C. Selecting Buffers for Perturbation

Given an instance $(\mathcal{A}_I, \mathcal{A}_F)$, candidate buffers are generated before the search. The process samples buffers overlapping with the fewest start and goal positions of objects, as well as of previously generated buffers. Besides the candidate buffers generated, unoccupied start and goal positions can also be used as buffers for some arrangement $\mathcal{A}_C$. Given this observation, each object o_i is assigned a buffer online by ranking candidates according to the following priorities. Buffers not overlapping with starts and goals are considered first as long as they are reachable from $\mathcal{A}_C[o_i]$. Then, start positions of objects already at goals in $\mathcal{A}_C$ are considered. Finally, goal positions not occupied in $\mathcal{A}_C$ are considered.

VII. Experiments

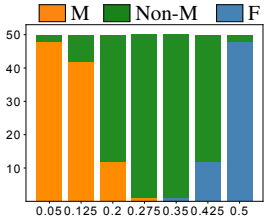

Fig. 4. A distribution of monotone (M), non-monotone (Non-M), and failed (F) cases among 50 instances (y-axis) for increasing density (x-axis).

To evaluate performance of the proposed algorithms, experiments on monotone and non-monotone instances are performed for different environment "density" levels. The density of the environment is defined as the ratio of the area occupied by objects to that of the environment. For a fixed number of objects, the density level can be increased by either increasing the object sizes or decreasing the workspace area. Fig. 4 shows the relationship between the probability of generating (or failing to generate) a monotone and non-monotone instance with 10 objects for different density levels. An attempt will fail when the environment has no more space for a valid start/goal placement. According to Fig. 4, the density level of the monotone and non-monotone problem should be chosen in the ranges $[0, 0.2]$ and $[0.125, 0.425]$ respectively.

A. Evaluation on Monotone Problems

For monotone problems, $\texttt{DFS}_{\texttt{DP}}$ is compared against several leading monotone solvers in the same field.

1) `mRS` (monotone rearrangement solver) - A backtracking method which searches over all possible orders with which the objects can be moved [8].
2) `fmRS` (fast-mRS) - Compute the sequence of moving objects via topological sorting on a constraint graph [12] constructed by assigning a transfer path per object.
3) `IP-Solver` - An integer programming (IP) solver which selects a path per object among all path options such that the corresponding dependency graph is acyclic.

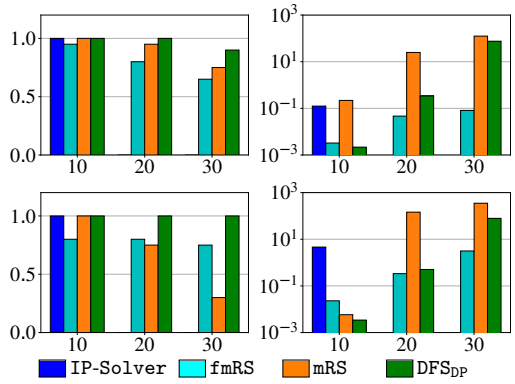

Fig. 5. [Top] Success rate (left) and computation time in seconds (right) with density level 0.1. [Bottom] The same evaluation with density level 0.2. Number of objects are shown on the x-axis.

The experiments are conducted in a square environment with density level 0.1 and 0.2 and the number of objects ranges between 10 and 30. For each setup, 20 instances are generated. A planning attempt is considered a failure if the method cannot determine monotonicity within 500 seconds or return a negative response. The success rate (left) and average running time (right) is plotted in Fig. 5.

The `IP-solver` has to pre-compute all the region graph paths for each pair of start and goal object positions, from which to make a choice, thus making it unscalable beyond 10 objects. `mRS` considers all possible moving orders (time complexity of $O(n!)$) while the proposed $\texttt{DFS}_{\texttt{DP}}$ keeps solving subproblems without considering the ordering of the solved subproblems ($O(2^n)$). Therefore, $\texttt{DFS}_{\texttt{DP}}$ significantly outperforms `mRS` both in higher success rate and lower computation time as the number of objects increases. The `fmRS` greedily assigns one path to each object with the minimum constraints violated, which indicates incompleteness compared to the complete $\texttt{DFS}_{\texttt{DP}}$. As a result, it is fast but tends not to find valid solutions in monotone instances.

B. Evaluation on Non-monotone Problems

For non-monotone problems, the focus is on cases where the optimal solution requires one or two buffers and can be

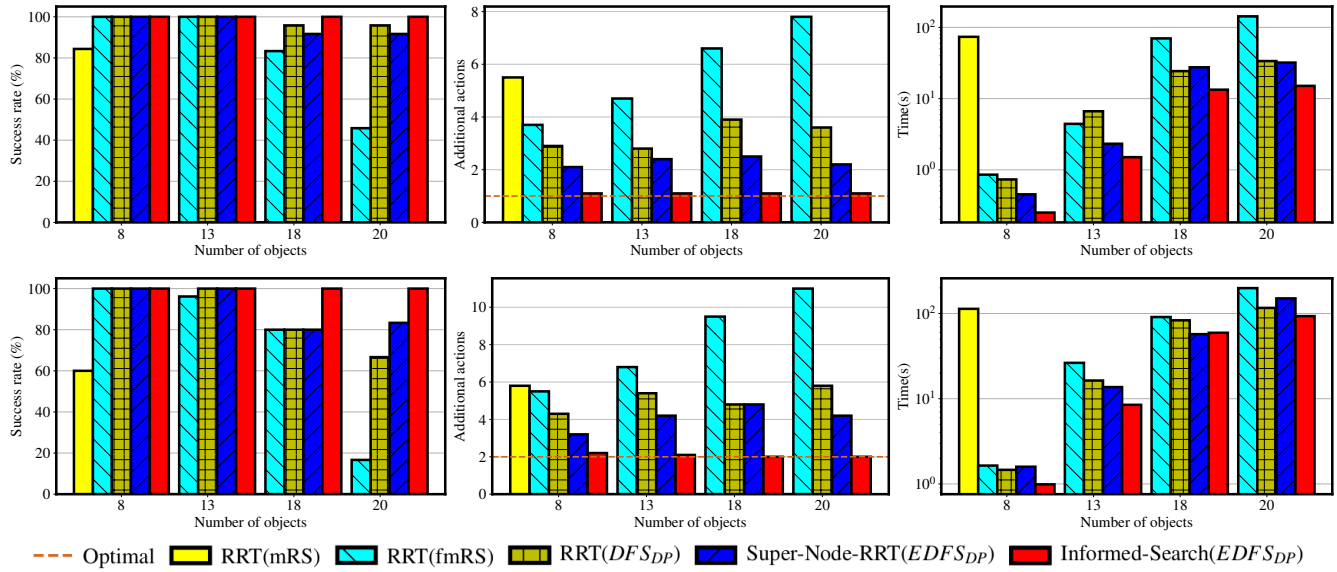

Fig. 6. Experimental results on one-buffer (top row) and two-buffer (bottom row) instances evaluating (1) success rate on finding a solution (left column) (2) additional actions needed to solve the problem (middle column) and (3) computation time (right column).

computed with an expensive, exhaustive brute force search approach. The density level is set to 0.225. The following comparison points are considered:

1) RRT(mRS) - Searches the arrangement space in an (RRT)-like fashion [31] and uses mRS as a local planner.
2) RRT(fmRS) - Same as above but fmRS as the local planner.
3) RRT(DFS_{DP}) - Same but DFS_{DP} as the local planner.
4) Super-Node-RRT(EDFS_{DP}) - Select the roots of super nodes for expansion and uses EDFS_{DP} as the local planner.
5) Informed-Search(EDFS_{DP}) - Same as above but uses heuristics to order super nodes, objects, and buffers.

Fig. 6 demonstrates the results for 1 and 2 buffer cases respectively in terms of success rate in finding a feasible solution in 300 seconds, number of buffers used and computation time. The two baselines RRT(DFS_{DP}) and Super-Node-RRT(EDFS_{DP}) search the arrangement space using an RRT-like process for sampling and therefore focus on finding feasible solutions, instead of high-quality ones. The total number of buffers is much higher than the final informed search pipeline (middle column) as the proposed informed search prioritizes super nodes with fewer number of perturbation to expand. In addition, the success rate starts to drop (left column) and the computation time starts to increase (right column) for the baselines as the number of objects increases, since the arrangement space increases exponentially and random perturbation node selection performance suffers. The improved baseline Super-Node-RRT(EDFS_{DP}) outperforms RRT(DFS_{DP}) by always selecting the root of super nodes and utilizing the benefits of EDFS_{DP} as a local planner. The proposed informed framework focuses the search to the most promising part of arrangement space and explores it more systematically. Therefore, it finds near-optimal solutions (middle column: 1.07 in one-buffer cases, 2.09 in two-buffer cases) with 100% success rate even in harder instances (left column: 18, 20 objects with 2 buffers). The computation time is lower than other methods in one-buffer cases and remains competitive in two-buffer cases.

The comparison results with RRT(mRS) and RRT(fmRS) are consistent with the observation made in monotone evaluation. The solution quality of RRT(fmRS) is worse than the baseline version RRT(DFS_{DP}) of the proposed search framework due to the incompleteness of the local solver. In addition, RRT(mRS) is not as scalable in non-monotone instances due to the weakness in quickly identifying non-monotonicity.

VIII. Conclusion and Future Work

This work tackles uniform object rearrangement with the goal of minimizing the total number of object movements. A region graph is introduced to decompose the configuration space and classify all continuous paths, upon which a complete and efficient monotone solver and then an effective informed search framework for non-monotone problems are proposed. The framework achieves high-quality solutions with high success rate and reduced computation time relative to alternatives.

The current study motivates future work for these efficient tools, including the extension to object-robot interactions and generalizations to non-uniform object geometries. Furthermore, the region graph can also be exploited to dynamically generate buffers, which adapts to specific arrangement environments to improve performance. It would be interesting to show (or disprove) that it is sufficient to consider a finite set of buffers for guaranteeing a complete approach. Machine learning methods can also be used to learn heuristics and object dependencies given access to solutions by this planning approach. This can lead to even more scalable solutions. Perception should also be taken into account to reason about pose hypotheses per object, which could affect the arrangement ordering, such as moving an object to make another more discernible. Different types of manipulation primitives can also be considered, such as non-prehensile actions (e.g., pushing, flipping) to enrich rearrangement strategies, which can be generalized to more sophisticated environments.

REFERENCES

[1] S. D. Han, S. W. Feng, and J. Yu, "Toward fast and optimal robotic pick-and-place on a moving conveyor," *IEEE Robotics and Automation Letters*, vol. 5, no. 2, pp. 446–453, 2019.

[2] S. D. Han, N. M. Stiffler, A. Krontiris, K. E. Bekris, and J. Yu, "Complexity results and fast methods for optimal tabletop rearrangement with overhand grasps," *The International Journal of Robotics Research*, vol. 37, no. 13-14, pp. 1775–1795, 2018.

[3] R. Shome, K. Solovey, J. Yu, K. Bekris, and D. Halperin, "Fast, high-quality dual-arm rearrangement in synchronous, monotone tabletop setups," *arXiv preprint arXiv:1810.12202*, 2018.

[4] G. Havur, G. Ozbilgin, E. Erdem, and V. Patoglu, "Geometric rearrangement of multiple movable objects on cluttered surfaces: A hybrid reasoning approach," in *2014 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2014, pp. 445–452.

[5] A. R. Dabbour, "Placement generation and hybrid planning for robotic rearrangement on cluttered surfaces," Ph.D. dissertation, 2019.

[6] R. M. Karp, "Reducibility among combinatorial problems," in *Complexity of computer computations*. Springer, 1972, pp. 85–103.

[7] A. Cosgun, T. Hermans, V. Emeli, and M. Stilman, "Push planning for object placement on cluttered table surfaces," in *2011 IEEE/RSJ international conference on intelligent robots and systems*. IEEE, 2011, pp. 4627–4632.

[8] M. Stilman, J.-U. Schamburek, J. Kuffner, and T. Asfour, "Manipulation planning among movable obstacles," in *Proceedings 2007 IEEE international conference on robotics and automation*. IEEE, 2007, pp. 3327–3332.

[9] J. van Den Berg, J. Snoeyink, M. C. Lin, and D. Manocha, "Centralized path planning for multiple robots: Optimal decoupling into sequential plans." in *Robotics: Science and systems*, vol. 2, no. 2.5, 2009, pp. 2–3.

[10] K. Hauser, "The minimum constraint removal problem with three robotics applications," *The International Journal of Robotics Research*, vol. 33, no. 1, pp. 5–17, 2014.

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

[12] ——, "Efficiently solving general rearrangement tasks: A fast extension primitive for an incremental sampling-based planner," in *2016 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2016, pp. 3924–3931.

[13] C. R. Garrett, T. Lozano-Pérez, and L. P. Kaelbling, "Ffrob: An efficient heuristic for task and motion planning," in *Algorithmic Foundations of Robotics XI*. Springer, 2015, pp. 179–195.

[14] 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)*. IEEE, 2014, pp. 639–646.

[15] O. Ben-Shahar and E. Rivlin, "To push or not to push," Computer Science Department, Technion, Tech. Rep., 1995.

[16] ——, "Practical pushing planning for rearrangement tasks," *IEEE Transactions on Robotics and Automation*, vol. 14, no. 4, pp. 549–565, 1998.

[17] E. Huang, Z. Jia, and M. T. Mason, "Large-scale multi-object rearrangement," in *2019 International Conference on Robotics and Automation (ICRA)*. IEEE, 2019, pp. 211–218.

[18] M. R. Dogar and S. S. Srinivasa, "A planning framework for non-prehensile manipulation under clutter and uncertainty," *Autonomous Robots*, vol. 33, no. 3, pp. 217–236, 2012.

[19] M. C. Koval, J. E. King, N. S. Pollard, and S. S. Srinivasa, "Robust trajectory selection for rearrangement planning as a multi-armed bandit problem," in *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2015, pp. 2678–2685.

[20] A. S. Anders, L. P. Kaelbling, and T. Lozano-Perez, "Reliably arranging objects in uncertain domains," in *2018 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2018, pp. 1603–1610.

[21] P. C. Chen and Y. K. Hwang, "Practical path planning among movable obstacles," Sandia National Labs., Albuquerque, NM (USA), Tech. Rep., 1990.

[22] 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.

[23] J. Van Den Berg, M. Stilman, J. Kuffner, M. Lin, and D. Manocha, "Path planning among movable obstacles: a probabilistically complete approach," in *Algorithmic Foundation of Robotics VIII*. Springer, 2009, pp. 599–614.

[24] H.-n. Wu, M. Levihn, and M. Stilman, "Navigation among movable obstacles in unknown environments," in *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE, 2010, pp. 1433–1438.

[25] Y. Kakiuchi, R. Ueda, K. Kobayashi, K. Okada, and M. Inaba, "Working with movable obstacles using on-line environment perception reconstruction using active sensing and color range sensor," in *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE, 2010, pp. 1696–1701.

[26] M. Levihn, M. Stilman, and H. Christensen, "Locally optimal navigation among movable obstacles in unknown environments," in *2014 IEEE-RAS International Conference on Humanoid Robots*. IEEE, 2014, pp. 86–91.

[27] M. R. Dogar, M. C. Koval, A. Tallavajhula, and S. S. Srinivasa, "Object search by manipulation," *Autonomous Robots*, vol. 36, no. 1-2, pp. 153–167, 2014.

[28] M. Danielczuk, A. Kurenkov, A. Balakrishna, M. Matl, D. Wang, R. Martín-Martín, A. Garg, S. Savarese, and K. Goldberg, "Mechanical search: Multi-step retrieval of a target object occluded by clutter," in *2019 International Conference on Robotics and Automation (ICRA)*. IEEE, 2019, pp. 1614–1621.

[29] C. Nam, J. Lee, Y. Cho, J. Lee, D. H. Kim, and C. Kim, "Planning for target retrieval using a robotic manipulator in cluttered and occluded environments," *arXiv preprint arXiv:1907.03956*, 2019.

[30] C. Nam, J. Lee, S. H. Cheong, B. Y. Cho, and C. Kim, "Fast and resilient manipulation planning for target retrieval in clutter," *arXiv preprint arXiv:2003.11420*, 2020.

[31] S. M. LaValle, "Rapidly-exploring random trees: A new tool for path planning," 1998.