Robot Packing With Known Items and Nondeterministic Arrival Order

Fan Wang, and Kris Hauser, *Senior Member, IEEE*

***Abstract*—This article formulates two variants of packing problems in which the set of items is known, but the arrival order is unknown. The goal is to certify that the items can be packed in a given container and/or to optimize the size or cost of a container so that that the items are guaranteed to be packable, regardless of arrival order. The nondeterministically ordered packing (NDOP) variant asks to generate a certificate that a packing plan exists for every ordering of items. Quasi-online packing (QOP) asks to generate a partially observable packing policy that chooses the location of each item as their arrival order is revealed one-by-one, with all of the remaining items certified to be packable regardless of their future arrival order. Theoretical analysis demonstrates that even the simple subproblem of verifying the feasibility of a packing policy is NP-complete. Despite this worst case complexity, practical solvers for both NDOP and QOP are developed. Multiple extensions to the basic nondeterministic problem are presented, including packing with a fixed-capacity buffer and packing with equivalent objects. Experiments demonstrate that these algorithms can be applied to packing irregular 3-D shapes with stability and manipulator loading constraints.**

***Note to Practitioners*—Automatic packing of objects into containers, such as a shipping box, has applications in warehouse automation for e-commerce applications and less-than-truckload shipping. In many scenarios, a container needs to be preselected for a set of objects before they arrive, and the order of the objects cannot be controlled. This article studies the theoretical foundations of packing problems, in which the set of items is known, but the arrival order is unknown. The algorithms presented in this article can certify whether a container can hold a set of objects in the case where the arrival order is revealed at once, one-by-one, and where a limited set of objects can be put aside in a buffer before packing. These algorithms can be used to choose optimal containers and packing strategies, both for robot and human packers.**

***Index Terms*—Bin packing, computational complexity, nondeterminism, robotics.**

Manuscript received March 25, 2020; revised August 4, 2020; accepted September 10, 2020. This work was supported by the Amazon Research Award. This article was recommended for publication by Associate Editor P. Burget and Editor K. Saitou upon evaluation of the reviewers' comments. *(Corresponding author: Kris Hauser.)*

Fan Wang is with the Department of Electrical and Computer Engineering, Duke University, Durham, NC 27708 USA (e-mail: fan.wang2@duke.edu).

Kris Hauser is with the Department of Computer Science, University of Illinois at Urbana–Champaign, Champaign, IL 61801 USA (e-mail: kkhauser@illinois.edu).

Color versions of one or more of the figures in this article are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TASE.2020.3024291

I. Introduction

INTEREST in warehouse automation has grown rapidly with the growth of e-commerce and advances in robotics. Given the rapid progress in the field of robotic manipulation, the prospect of fully autonomous picking and packing robots is becoming increasingly likely in the near future [1], but relatively little attention has been paid to robotic packing. Packing algorithms have the potential to optimize containers and packing plans for both human and robot packers. In the current state of practice in fulfillment centers, human workers select containers and pack items largely according to intuition. Heuristic algorithmic assistance based on item bounding box dimensions may be employed, but these lead to conservatively large containers. When containers are chosen too small, items need to be repacked, causing delays and reducing efficiency. When containers are too large, excess material is wasted, and shipping costs are increased.

A large variety of packing problems have been studied, including the bin and strip packing problem, knapsack problem, container loading problem, nesting problem, and others. In an *offline* setting, the items and container(s) are known, and a plan can place the items in arbitrary order [2]. In an *online* setting, the items are not known *a priori* and need to be placed as they arrive [3]. We consider a *robot packing* setting, which addresses packing problems with the additional constraints that items must be loaded with a collision-free robot path and that intermediate piles of items must be stable against gravity.

This article introduces two *nondeterministic* formulations of robot packing problems that lie between the offline and online settings. These formulations are practical for automated warehouses where the ultimate item set (e.g., shopping cart) is known, but some distinct, uncontrollable component of the packing system controls the item arrival order. For example, in Amazon's automated fulfillment centers, shelving units containing individual items are carried by thousands of mobile robots to several picking stations, and the order in which shelves arrive at a given station is controlled by a complex algorithm that is tuned to maximize delivery throughput for shelving units. In applications where item deliveries are human-controlled, such as in less than truckload consolidation, it may be even less practical for an algorithm to dictate the arrival order. Hence, to guarantee that the items can fit in a given container, a packing planner should *certify* the validity of a plan under *all possible arrival orders*. In the nondeterministically ordered packing (NDOP) variant, the feasibility of the container is verified under all nondeterministic orders,

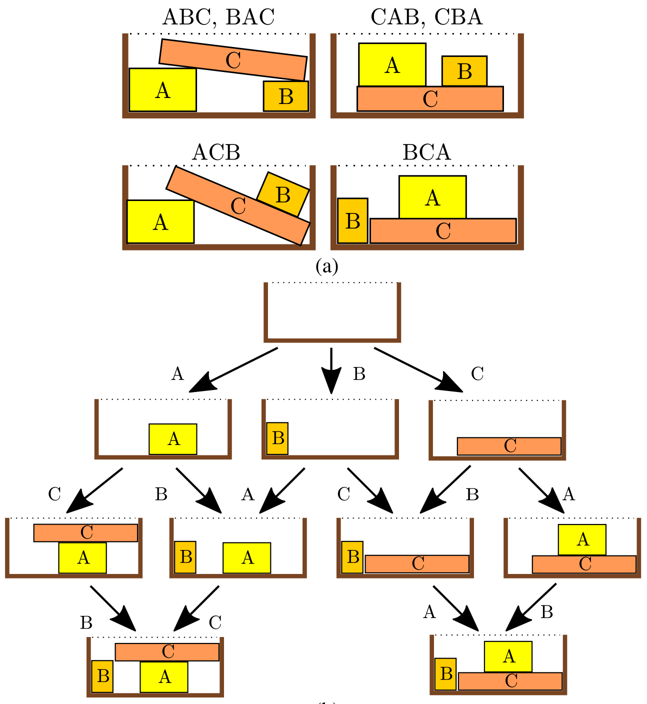

Fig. 1. Feasible solutions for a 2-D, three-item instance of (a) NDOP and (b) QOP. All $3! = 6$ possible arrival orders are collision-free, loadable from top-down, and yield intermediate piles that are stable under gravity. In QOP, an item is never moved after it is placed. (a) NDOP. (b) QOP.

but the arrival sequence is revealed before packing is executed. In the quasi-online packing (QOP) variant, each object must be packed before the next item is revealed (Fig. 1).

Our framework for solving NDOP and QOP problems uses a combination of an offline planner and a packing policy verifier. A packing policy is represented by a set of possible packing plans, each of which consists of a set of packing locations and a directed acyclic graph (DAG) of their dependencies. The offline planner is treated as an *oracle* that is supplied independent of the NDOP/QOP algorithm and is assumed to handle all geometric and physical constraints required of the packing domain. The verifier will verify or disprove the feasibility of a policy under all permutations of arrival orders. We present a verification algorithm that uses pruning techniques, and in practice, can check feasibility quickly even for a large number of objects and packing plans. However, in some cases, exponential behavior is observed. We prove that the worst case solution complexity of NDOP and QOP is $O(n!)$, and even feasibility verification for a polynomial-sized NDOP policy is NP-complete via a reduction from Boolean satisfiability (SAT).

Nevertheless, the solver is practical for small numbers of items and even using an incomplete offline planner; it guarantees that a solution, when found, is feasible for all object orderings. Several packing heuristics are also introduced to improve the scalability of the approach, and experiments demonstrate that our approach can be realistically applied to irregular 3-D shapes with item sets of size up to 10.

This article is an extended version of a previous conference publication [4]. This extended version presents two novel extensions to the prior work. First is a modification to the basic NDOP and QOP algorithms to handle item equivalence classes, which can improve scalability when orders contain multiples of a given item (Section VII). Second is a new algorithm for the problem of k-buffered nondeterministic packing, which is applicable when a packing setup contains an auxiliary space or multiple hands that can hold up to k objects (Section VIII). Experiments on ten-object item sets in both of these new settings demonstrate that the awareness of equivalence classes and the addition of buffers both improve running times significantly. Moreover, larger buffers allow for denser packing in the worst case, which has consequences for workcell design.

II. Related Work

Packing algorithms have been studied extensively both for their theoretical interest and practical applications in shipping, manufacturing, and 3-D printing. The vast majority of work considers rectilinear objects. State-of-the-art exact algorithms for the offline 2-D and 3-D bin packing problem use branch-and-bound approach [2], [5]. Because exact methods have worst case exponential complexity, heuristic methods and metaheuristic approaches have been developed, such as the bottom-left [6] and best-fit-decreasing heuristics [7]. Heuristics are the only practical methods available for irregular shape packing (also known as nesting [8]) since the freedom to rotate leads to continuously infinite search space. Metaheuristic optimization methods [9] simultaneously optimize the placements of all items, and constructive heuristics incrementally place items according to some scoring function [10], [11]. Due to recent advances in systems for warehouse automation [1], [12], researchers have been turning to new considerations that arise when implementing packing in robotic workcells. Shome *et al.* [13] present a system for packing identical rectilinear objects that uses vision feedback and corrective manipulations to address errors that occur during tight packing. den Boef *et al.* [14] formulate constraints to avoid collisions with a robot with a vacuum gripper during packing. Wang and Hauser [15] formulate pile stability and manipulation feasibility constraints in a packing planner that can handle arbitrary, nonconvex objects. Similar types of constraints have been considered in research that has extended classical assembly planning approaches to physical robot manipulators [16].

In the offline packing setting, the planner assumes that the item set and packing order are controlled. Most classical versions do not formulate interdependence between items (i.e., items appear and then "float" in their planned locations), which means ordering is irrelevant. This is a reasonable assumption for some scenarios, such as sheet metal cutting, but one should consider additional constraints when packing containers in practice. Prior work has enforced clearance of boxes along with axis-aligned loading directions in 3-D bin-packing by ensuring no previously placed item lies along an extruded prism along at least one face of the box [14]. Recent work has also formulated collision checking between the loading mechanism (human hand, forklift, robot hand, and so on) and already-placed items [15]. Stability constraints [10], [15], [17] also impose dependence on the packing order. Due to these dependencies, if the item arrival order does not match the planned order, the plan might not be successfully executed.

In the online setting, an arbitrary item is presented to the algorithm, which then chooses a packing location [3]. Neither the item set nor the packing order is controllable, and often the item geometry can also be arbitrary as well. There are no guarantees that a given container can be packed, and therefore, the typical formulation casts the problem as an optimization of the number of containers or the container height. Competitive ratios are known for various online algorithms in the 1-D and 2-D rectilinear bin packing settings [3], [18], but to the best of our knowledge, no results are known for irregular shapes.

In contrast, NDOP and QOP are two novel points on the spectrum between offline and online packing. They allow for greater uncertainty than the offline setting, but seek guaranteed packing in a single container when the item set is known, which is more appropriate for fulfillment applications than the online setting. Our NDOP and QOP solvers can handle irregular shapes, and we prove that when they successfully compute a policy, the policy correctly packs the given items regardless of arrival order.

III. Glossary of Symbols and Terms

A_i	the geometry of the item with index i
B	a set of items in a buffer
CDG	constraint dependency graph
G	a directed graph
$\mathcal{G}$	a set of graphs
$\mathcal{I}$	the set of items
$\tilde{\mathcal{I}}$	the set of classes of unique items
item i	shorthand for “the item with index i”
n	item count
N	a node in a policy tree
NDOP	nondeterministically ordered packing
QOP	quasi-online packing
P	an offline packing plan
$\mathcal{P}$	a set of offline packing plans
π	a packing policy
S_n	the set of permutations on n elements
SE(3)	the set of three dimensional rigid transforms
σ_i	index of the i'th item in a packing sequence
$\sigma_{1:j}$	a packing sequence of j items $\sigma_1, \ldots, \sigma_j$
SV	a swept volume
T_i	the rigid transform of the item with index i
v_i	the item in $\mathcal{I}$ with index i

IV. Problem Formulation

Let $\mathcal{I} = \{v_1, \ldots, v_n\}$ be a set of n items. Item v_i has some geometry $A_i \subset \mathbb{R}^d$, and we wish to pack all items into a container volume $C \subset \mathbb{R}^d$. Here, $d = 2$ or 3 is the dimension of the workspace. Since each item is uniquely associated with its index, throughout this article we often refer to “item i” rather than “item v_i”. The offline packing problem is to compute a *feasible packing plan* given $A_1, \ldots, A_n$ and C. Such a plan is defined as follows.

Definition 1: A *packing plan* P consists of an ordering $\sigma_{1:n} = (\sigma_1, \ldots, \sigma_n)$ and a tuple of transforms

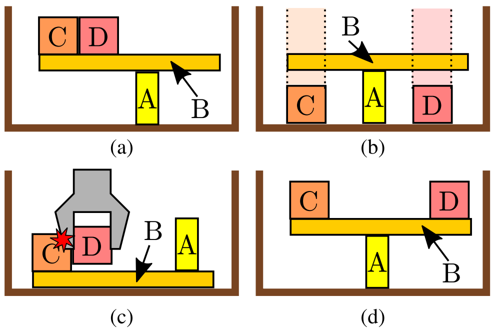

Fig. 2. Examples of plans that are infeasible for arrival order ABCD: (a) unstable, (b) items C and D collide with B along the loading direction, and (c) path for the robot manipulator to grasp and load item D is infeasible. (d) although ABCD is feasible, the prefix requirement is violated because the subplan ABC is unstable.

$T_{1:n} = (T_1, \ldots, T_n)$, in which $\sigma_j \in \{1, \ldots, n\}$ specifies that v_{σ_j} is the jth item to be placed, and $T_i \in SE(d)$ specifies the target location (pose) of v_i.

Ordering must be a permutation on n elements, and is, hence, an element of the symmetric group S_n.

Definition 2: A packing plan is *feasible* when it, and all prefix plans, satisfy certain constraints, as shown in Fig. 2 and defined in Section IV-A.

The prefix feasibility requirement means that for all $j < n$, the ordering $\sigma_{1:j}$ with the corresponding items in locations $T_{\sigma_1}, \ldots, T_{\sigma_j}$ must also satisfy the feasibility constraints. For example, we cannot require two blocks to be placed simultaneously on either ends of a see–saw when stability is violated with only a single block [Fig. 2(d)].

A. Constraint Formulation

In our formulation, the feasibility of a packing plan requires satisfying the following three constraints. For readability, for the ordering $\sigma_{1:n}$, let us denote the sequence number s_i of the ith item to be the ordinal index in which it appears, i.e., $s_{\sigma_j} = j$ and $\sigma_{s_i} = i$.

a) Noninterference: All items do not overlap but can touch (1) and all items lie entirely inside the container (2)

$$T_i A_i^\circ \cap T_j A_j^\circ = \emptyset \text{ for all } i, j \text{ with } i \neq j \tag{1}$$

$$T_i A_i \subseteq C \text{ for all } i. \tag{2}$$

Here, $\cdot^\circ$ denotes a set's interior.

b) Equilibrium: To prevent “floating” items and unbalanced stacks, the equilibrium constraint requires that each intermediate packing be stable under gravity and frictional contact [Fig. 2(a)]. An item is allowed to make contact with the container walls and previously placed items. We model these as a set of contact points and require that there exist feasible forces at each contact point that respect Coulomb friction while maintaining torque and force balance. The set of contacts is determined by finding all nearby triangles between pairs of objects within a given margin ϵ (set to 2.5 mm in our experiments), generating the closest points between triangles, and eliminating duplicate points. We check the equilibrium conditions by solving a convex program [19].

c) Manipulation feasibility: Each item in the packing plan must be loadable by a manipulator without disturbing previously packed items [Fig. 2(b) and (c)]. We consider a robot gripper R and a top-down loading direction. In the packing plan, an item is also given a grasp transform T_G, such that the combined geometry of the ith item and the robot while grasped is $A_i \cup T_G R$. The swept volume of the item and robot while loading is $SV_i = \overline{ab} \oplus T_i(A_i \cup T_G R)$, where $\oplus$ denotes the Minkowski sum, and $a = (0, 0, 0)$ and $b = (0, 0, h)$, with h some "safe" height greater than the height of the container. This constraint states that, for all items i, the swept volume cannot intersect any previously placed items (3) or the container walls $W \subset \partial C$ (4)

$$SV_i^\circ \cap T_j A_j = \emptyset \text{ for all } j \text{ s.t. } s_j < s_i \tag{3}$$
$$SV_i^\circ \cap W = \emptyset. \tag{4}$$

In practice, we check this constraint approximately by dividing the path in small steps and performing collision checking for each discretized pose of the item and robot.

B. Nondeterministic Problems

Definition 3 (NDOP): The *NDOP problem* asks whether there exists a feasible packing plan for every ordering $\sigma_{1:n} \in S_n$.

To define QOP, we need to define the concept of a *feasible packing policy* as follows.

Definition 4: A *packing policy* is a function π that takes as arguments the identities and locations of previously packed items $(T_{\sigma_1}, \ldots, T_{\sigma_{j-1}})$ and the next item σ_j to be packed, and returns the location T_{σ_j} of the next packed item.

Definition 5: A packing plan is *generated* by a packing policy π and an ordering $\sigma_{1:n} \in S_n$ via the recursive application of the policy

$$\begin{aligned} T_{\sigma_1} &= \pi((), \sigma_1) \\ T_{\sigma_2} &= \pi((T_{\sigma_1}), \sigma_2) \\ &\vdots \\ T_{\sigma_n} &= \pi((T_{\sigma_1}, \ldots, T_{\sigma_{n-1}}), \sigma_n). \end{aligned} \tag{5}$$

Definition 6: A *packing policy is feasible* if for all item orders $\sigma_{1:n} \in S_n$, the generated packing plan is feasible.

Since a packing policy is deterministic, we can also write the policy as a function of the prior order of the objects

$$\pi((\sigma_1, \ldots, \sigma_{j-1}), \sigma_j) \equiv \pi((T_{\sigma_1}, \ldots, T_{\sigma_{j-1}}), \sigma_j). \tag{6}$$

A policy can also be viewed as a tree with depth n and each node has $n - \ell$ branches on level ℓ. There are $n \cdot (n-1) \cdots (n-\ell)$ nodes on level ℓ, and therefore, this gives a total of $\sum_{\ell=1}^{n} n!/\ell! = O(n \cdot n!)$ nodes altogether.

Definition 7 (QOP): The *QOP problem* asks to compute a feasible packing policy.

The main difference between NDOP and QOP is that with QOP, the items are revealed in sequence, and the location chosen for an item is fixed and may not be changed thereafter. QOP is at least as hard as NDOP because any solution to QOP is also a solution to NDOP (but the converse does not hold).

Limitations: It is important to note that our algorithms assume that when a packing plan determines a target pose for an item, the robot will be able to load the item as desired. That is, we do not reverify whether the object in the arrival pose is graspable as planned. In some problem settings, this assumption may be violated, such as when the robot wishes to pack a wine glass holding it from the top, but the glass arrives facing down. A partial solution is to replan a new grasp when the item arrives such that it is geometrically compatible with the plan. This is likely to succeed when arrival poses uncertainty is small, or the items are symmetric. A better, complete solution would be to calculate a regrasping strategy to place the object in the desired orientation so that it can be packed successfully [20]. Another assumption is that the plan is executed perfectly. In real-world scenarios, errors in geometric measurement, grasping friction, mass, and center of mass estimation may cause execution to deviate from the plan. We leave the problems of nondeterministic packing with uncertainty in arrival pose (without regrasping) and uncertainty in execution as open problems for future work.

C. Container Optimization Variants

Earlier, we have stated these packing problems in their decision versions. We also consider container optimization variants, which assume a set of possible containers $\mathcal{C}$ and a cost function $cost : \mathcal{C} \to \mathbb{R}$, and are stated as follows.

1) Offline: Find the container $C \in \mathcal{C}$ with a minimum cost that yields a feasible packing plan for item set $\mathcal{I}$.
2) Nondeterministically ordered: Find the container $C \in \mathcal{C}$ with a minimum cost that yields a feasible packing plan for any ordering of item set $\mathcal{I}$.
3) Quasi-online: Find the container $C \in \mathcal{C}$ with a minimum cost that yields a feasible packing policy for item set $\mathcal{I}$.

The container set is typically discrete, such as a set of available boxes, but could also be continuous, such as a varying height. This formulation can express the classical bin-packing problem, where $\mathcal{C}$ contains a container with one bin, a container with two bins, and so on, and cost measures the number of bins.

NDOP and QOP are adapted rather easily into discrete container optimization algorithms by enumerating containers in order of nondecreasing cost until a successful packing policy is found.

V. Method

We make use of an offline robot packing planner [15] with a small modification. The responsibility of the offline planner is to generate a feasible packing plan given the constraints outlined earlier, while our key contributions are novel methods to invoke the planner and to validate plans under permutations of item orders. Our NDOP and QOP algorithms treat the offline planner as an oracle that is responsible for all constraint checking and dependence graph construction steps. As a result, the NDOP and QOP algorithms can be generalized to other packing settings by replacing the oracle with other methods.

A. Offline Planning Oracle

The offline planner is required to accept some number of fixed items and a partial packing sequence for the remaining items. Its interface takes the form

$$P \leftarrow \text{Offline-Pack}\big(\sigma^{\text{fixed}}_{1:j}, P^{\text{prior}}, \sigma^{\text{next}}_{j+1:k}\big) \tag{7}$$

producing either a feasible plan $P = (\sigma_{1:n}, T_{1:n})$ or "failure." The inputs $\sigma^{\text{fixed}}_{1:j}$ specify that j items of the prior plan P^{prior} should be kept in their previous positions, and $\sigma^{\text{next}}_{j+1:k}$ are a sequence of $k - j > 0$ items that should be placed next. The remaining $n - k$ items can be placed in arbitrary order. Specifically, the result must satisfy $\sigma_{1:j} = \sigma^{\text{fixed}}_{1:j}$, $\sigma_{j+1:k} = \sigma^{\text{next}}_{j+1:k}$, and each fixed transform T_j for $j \in \sigma^{\text{fixed}}_{1:k}$ matches the corresponding transform in P^{prior}.

The offline planner used here is a constructive, heuristic method that is easily modified to handle the required changes. In Section VI, we also consider offline packing heuristics that make it easier to generate a nondeterministic plan, but these are not strictly necessary to ensure the completeness of our method.

B. Compatibility

A naive algorithm to solve NDOP would compute a packing plan for all $n!$ orderings. However, the notion of *plan compatibility* allows us to validate large numbers of orderings for lightly interdependent plans. For example, if we ignored manipulation feasibility and equilibrium constraints, there is no sequential dependence between any two items, and hence, all orderings of items would be feasible under the following policy: *when an item arrives, just place it in its planned location.* We define compatibility as follows.

Definition 8 (Compatible ordering): A packing plan $P = (\sigma_{1:n}, T_{1:n})$ is *compatible with an ordering* $\sigma'_{1:n}$ if the reordered plan $P' = (\sigma'_{1:n}, T_{1:n})$ is feasible.

Hence, we can recast the problem of generating a feasible packing policy as one of generating a set of feasible plans with sufficient coverage as follows.

Definition 9 (NDOP #2): Compute a set of feasible plans $P_1, \ldots, P_m$ such that for any order $\sigma_{1:n} \in S_n$, there is at least one plan compatible with $\sigma_{1:n}$.

Our NDOP solver formulates a packing policy as a set of packing plans $P_1, \ldots, P_m$ along with their associated *constraint dependence graphs* (CDGs) $G_1, \ldots, G_m$ as defined in Section V-C. An individual packing plan can be used for the set of orderings that are compatible with its dependence graph. If the union of the m sets of compatible orderings covers S_n, then we are done. If not, we find an incompatible ordering using Algorithm 1 and generate a new plan for this ordering.

A QOP solver must address the problem that if any two plans share the same order prefix, each item location in the prefix must be the same. Our algorithm uses the same CDG data structure to calculate compatibility while generating an optimized policy tree.

C. Constraint Dependence Graphs

A fundamental data structure that will allow us to verify compatibility is the constraint dependence graph (CDG). This

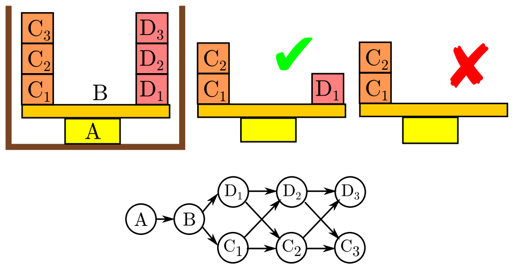

Fig. 3. Plan and its dependence graph. C_2 requires D_1 to be present to maintain the stability constraint, because, otherwise, the imbalanced weight on B would cause tipping. Similarly, D_2 depends on C_1, and so forth for C_3 and D_3. This CDG is compatible with orders of the form $AB(C_1\ D_1)(C_2\ D_2)(C_3\ D_3)$, where the (XY) denotes either XY or YX.

structure (Fig. 3) explicitly models the dependencies between items, so that compatibility can be quickly verified.

Definition 10 (CDG): The CDG of a feasible plan P is a graph on vertices $\mathcal{I}$ that has an edge (u, v) if some feasibility constraint requires item u to be placed before item v.

We can see that a CDG is a DAG because if there were a cycle in the graph, by transitivity, an item on the cycle would need to be placed before itself. Moreover, a CDG can be replaced by its transitive reduction with no loss in compatibility information.

To construct a CDG $G = (\mathcal{I}, E)$ of a plan $P = (\sigma_{1:n}, T_{1:n})$, we do so in incremental fashion by testing all pairwise constraints. Observe that there is no edge $(\sigma_j, \sigma_i) \in E$ for $i < j$, and we need not add edges (u, σ_i) for any ancestors of σ_i already in the CDG. For each index i in increasing order, we check all $u \in \sigma_{1:i-1}$ for a dependence in reverse packing order. First, if u is an ancestor of σ_i, it is skipped because σ_i is already dependent on u. Next, the manipulation feasibility constraint of u is checked against σ_i. If so, we add an edge (u, σ_i). If not, we proceed to check the equilibrium of the partial stack that includes $\sigma_{1:i}$ but omits u and all descendants of u. If there is no equilibrium solution, we add an edge (u, σ_i) (see Fig. 3). It should be noted that there exist scenarios that are stable if a single predecessor item is removed, but unstable if multiple predecessors are removed. These examples, however, are convoluted "multiple see–saw" constructions and would be highly unlikely to be generated by an offline packing planner.

An ordering is compatible with a plan P if it does not violate any dependence in P's CDG. In other words, a feasible packing plan P with a dependence-free CDG $G = (\mathcal{I}, \emptyset)$ is compatible with all orderings. More precisely, we can state the following.

Lemma 1: Let $G = (\mathcal{I}, E)$ be the CDG of a feasible plan P. An ordering $\sigma'_{1:n}$ is incompatible with P if there exists indices $u < v$ such that $(\sigma'_v, \sigma'_u) \in E$.

In other words, a compatible ordering obeys all pairwise ordering constraints specified by the edges of the CDG.

D. Coverage Verification

A key subroutine used in our NDOP and QOP planners verifies whether a set of packing plans $P_1, \ldots, P_m$ is compatible with all orderings in S_n, and if not, to generate a

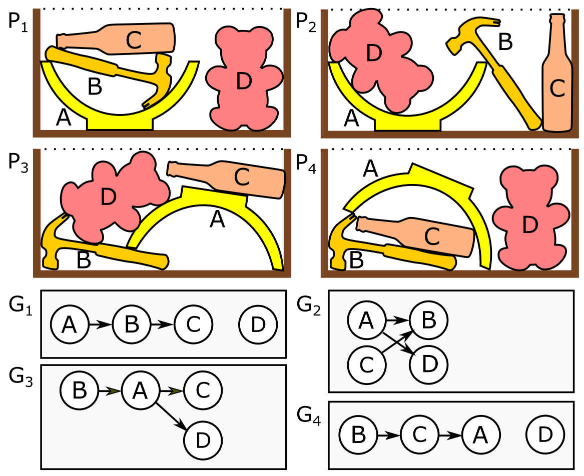

Fig. 4. Set of plans $P_1, \dots, P_4$ (top) and their dependence graphs $G_1, \dots, G_4$ (bottom). For any ordering beginning with A, there is at least one plan (P_1 or P_2) compatible with it. But for any ordering beginning with BDA, CB, CD, DAC, or DC, no plans are compatible.

counterexample (i.e., incompatible ordering). An example is shown in Fig. 4. Let us reduce this to a combinatorial problem of validating whether a set of dependence graphs is compatible with all orderings, and call it DEPSET-COMPAT.

We present a recursive algorithm, which tries assigning each unassigned vertex v, and recurses on the subset of plans in which v is a root. There are two base cases.

1) There exists a vertex v that is not a root in any G_i. Then, any order that starts with v is a counterexample.
2) G_i has no edges for some plan P_i. The policy is feasible because P_i is compatible with all orderings.

To verify faster, we also perform a *singleton pruning* step: if a vertex v is a singleton (has no neighbors) in every $G_1, \dots, G_m$, then v can be safely ignored. This is because v can be assigned at any point without affecting dependencies.

Algorithm 1 Verify-CDG-Coverage($\mathcal{I}, \mathcal{G}$)

```
   input : a set of items $\mathcal{I}$
           list of dependency graphs $\mathcal{G} = (G_1, \dots, G_m)$
1  if there exists $v \in \mathcal{I}$ that is not a root in any graph $G_i$
   then return "$v$ incompatible";
2  if any $G_i \in \mathcal{G}$ has no edges then return "all
   compatible";
3  Remove all vertices $v$ from $\mathcal{I}$ that are singletons in every
   graph $G_i \in \mathcal{G}$;
4  for $v \in \mathcal{I}$ do
5      $\mathcal{G}^v \leftarrow ()$;
6      for $i = 1, \dots, m$ do
7          if $v$ is a root in $G_i$ then
8              Append $G_i$ to $\mathcal{G}^v$, but with $v$ removed;
9      end
10     $r \leftarrow$Verify-CDG-Coverage($\mathcal{I} \setminus \{v\}, \mathcal{G}^v$);
11     if $r =$"$\sigma_{1:j}$ incompatible" then
12         return "$v, \sigma_{1:j}$ incompatible"
13 end
14 return "all compatible"
```

The overall algorithm is given a vertex set $\mathcal{I}$ and a list of CDGs $\mathcal{G} = (G_1, \dots, G_m)$ of the CDGs of $P_1, \dots, P_m$ as input, and is listed in Algorithm 1. The return value is either "all compatible" or a subsequence of $\mathcal{I}$ that is incompatible with every dependence graph. Line 1 processes the first base case, and line 3 processes the second. Line 3 performs the singleton pruning step, and lines 4–14 perform the recursion. Lines 5–10 compute the list $\mathcal{G}^v$ of dependence graphs that are compatible with assigning v at the current step, but with v is removed. In line 13, the vertex v is prepended to the counterexample of a recursive call because the counterexample is reached after assigning v. (We note that instead of copying CDGs in lines 5–10, it is more efficient to modify the CDGs in-place and then undo the operations after line 12.)

A counterexample can often be found faster by ordering the vertices in line 4 using a heuristic. Our approach sorts the vertices v by the number of plans compatible with the assignment of v (i.e., have v as a root).

Algorithm 2 solves NDOP (version #2) using the Verify-CDG-Coverage subroutine.

Algorithm 2 NDOP-Solve

```
1  r=    input : a set of items $\mathcal{I}$
   output: a solution set of plans $\mathcal{P}$, or "failure"
1  $\mathcal{E}$ ←empty-list;
2  $\mathcal{P}$ ←empty-list;
3  while true do
4      $r \leftarrow$ Verify-CDG-Coverage($\mathcal{I}, \mathcal{E}$);
5      if $r =$"all compatible" then return $\mathcal{P}$;
6      Let $\sigma_{1:\ell}$ be the incompatible ordering in $r$;
7      $P \leftarrow$Offline-Pack($nil, nil, \sigma_{1:\ell}$);
8      if $P =$"failure" then return "failure";
9      Add $P$ to $\mathcal{P}$;
10     Add $CDG(P)$ to $\mathcal{E}$;
11 end
```

E. Quasi-Online Packing

Due to the need for shared transforms, QOP is not as amenable to the elimination of orderings via compatibility verification. A naïve method for QOP would build a policy tree by enumerating all possible orders and ask for compatible plans.

Specifically, let N be a node in the policy tree at depth ℓ, which is associated with the ℓth step of a feasible plan $P = (\sigma_{1:n}, T_{1:n})$. For all nonplaced items $\sigma'_{\ell+1} \notin \sigma_{1:\ell}$, we could call

$$P' \leftarrow \text{Offline-Pack}(\sigma_{1:\ell}, P, \sigma'_{\ell+1}). \quad (8)$$

If P' ="failure," then failure is returned. Otherwise, P' is associated with a new child of N in the tree corresponding to the choice $\sigma'_{\ell+1}$, and the search can proceed recursively. Note that if $\sigma_{\ell+1}$ was already the $\ell + 1$th item in P, replanning is unnecessary and we can just set $P' = P$. With this check, only $O(n!)$ calls to the offline planner are needed.

This procedure can be optimized by observing that all items that are roots of the dependence subgraph

$CDG(\sigma_{\ell+1:n}, (T_{\sigma_{\ell+1}}, \ldots, T_{\sigma_n}))$, can reuse P. In fact, all *combinations of roots* can reuse P. Moreover, once roots have been assigned, any newly created children can also reuse it.

To exploit this, our QOP planner performs a depth-first search according the pseudocode given in Algorithm 3, which is invoked using Algorithm 4. Each search node N at level ℓ is associated with

1) a packing sequence $\sigma_{1:\ell}$;
2) a transform T_{σ_ℓ} for item σ_ℓ;
3) a list of plans $\mathcal{P}_N$ compatible with $\sigma_{1:\ell}$ (i.e., all T_j match, for each fixed $j \in \sigma_{1:\ell}$).

Each plan $P_i \in \mathcal{P}_N$ is associated with a dependence subgraph G_i, which is the subgraph P_i's CDG induced by the unpacked vertices $\{\sigma \notin \sigma_{1:\ell}\}$. Line 3 gives the single-layer packing base case that allows early termination. Lines 5–16 proceed in depth-first search fashion to enumerate children of N. For each item arrival $\sigma_{\ell+1}$, a child node C is generated. If at least one plan in $\mathcal{P}$ is compatible with $\sigma_{\ell+1}$, i.e., that $\sigma_{\ell+1}$ is a root of some G_i (lines 6 and 7), then replanning is not performed and the choice $T_{\sigma_{\ell+1}}$ is fixed. If multiple plans are compatible, the value of $T_{\sigma_{\ell+1}}$ that is compatible with the most plans is used. $\mathcal{P}_C$ is then set to the set of plans in $\mathcal{P}_N$ for which item $\sigma_{\ell+1}$ is a root of the dependence subgraph, and whose placement of item $\sigma_{\ell+1}$ matches $T_{\sigma_{\ell+1}}$. If no plan is compatible (lines 9–13), then offline-pack is called as normal, and $\mathcal{P}_C$ is set to contain only the newly generated plan P'. Moreover, we add P' to the sets $\mathcal{P}_A$ for any ancestor of N (inclusive), which enables subsequent siblings, siblings of parents, and so on, to use P' and avoid additional planning (line 13).

Algorithm 3 QOP-Recurse(N)

```
   input : policy tree node $N$ at depth $\ell$
1  Let $\sigma_{1:\ell}$ be the sequence of packed items in $N$;
2  Let $\mathcal{P}_N$ be the set of plans in $N$;
3  if for any $P \in \mathcal{P}_N$ the subgraph of $P$ induced by
     $\{\sigma \notin \sigma_{1:\ell}\}$ has no edges then return "success";
4  for all items $\sigma_{\ell+1} \notin \sigma_{1:\ell}$ do
5      if any plan in $\mathcal{P}_N$ is compatible with $\sigma_{1:\ell+1}$ then
6          Let $T_{\sigma_{\ell+1}}$ be the location compatible with the most
           plans in $\mathcal{P}_N$;
7          $\mathcal{P}_C \leftarrow \{P' \in \mathcal{P}_N | \quad P'$ is compatible with $T_{\sigma_{\ell+1}}\}$;
8      else
9          Let $P$ be any plan in $\mathcal{P}_N$, or $nil$ if $\mathcal{P}_N = \emptyset$;
10         $P' \leftarrow$Offline-Pack($\sigma_{1:\ell}, P, \sigma_{\ell+1}$);
11         if $P' =$"failure" then return "failure";
12         $\mathcal{P}_C \leftarrow \{P'\}$;
13         For all ancestors $A$ of $N$, add $P'$ to $\mathcal{P}_A$;
14     end
15     $C \leftarrow$ add-child($N, \sigma_{\ell+1}, \mathcal{P}_C$);
16     if QOP-Recurse($C$) fails then return "failure";
17 end
18 return "success"
```

F. Analysis

Here, we show that NDOP-Solve (Algorithm 2) inherits the completeness properties of the offline planner, but QOP-Solve

Algorithm 4 QOP-Solve()

```
1 $root \leftarrow$ make-node($nil, \emptyset$);
2 if QOP-Recurse(root) is successful then return $root$;
3 else return "failure";
```

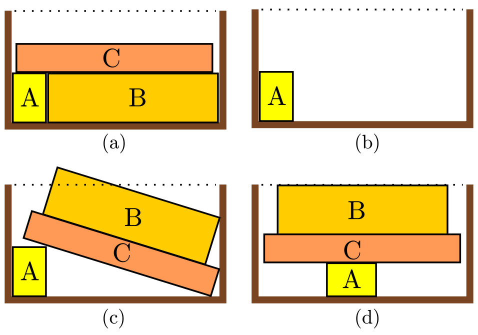

Fig. 5. Example showing that QOP-Solve (Algorithm 4) is not necessarily complete even with a complete offline planner. (a) First recursive call produces a feasible plan with A placed first. (b) Once item A is placed in the planned location, the plan is infeasible for order ACB, as shown in (c). On the other hand, if A were placed as in (d), a feasible QOP solution could be found.

(Algorithm 4) is incomplete. Even when the offline planner is incomplete, when the NDOP-Solve or QOP-Solve result is not "failure," the solution is correct. We also analyze the behavior of Verify-CDG-Coverage (Algorithm 1) and demonstrate that it is NP-complete.

1) Correctness and Completeness: NDOP-Solve inherits its completeness from the offline packing planner. To see this, first observe that whenever Algorithm 2 returns a solution (line 5), this solution is correct, because for all orderings $\sigma_{1:n} \in S_n$, Algorithm 1 has shown that the solution contains some plan that is compatible with $\sigma_{1:n}$. Now, consider the case where NDOP-Solve returns "failure." This can only occur when the failure of offline-pack returns for a partial ordering $\sigma_{1:j}$ (line 8). If the offline-pack is complete, then there is indeed no solution compatible with this ordering, and hence, NDOP-Solve returns failure correctly. If it is incomplete, then NDOP-Solve may return failure incorrectly.

Assuming Verify-CDG-Coverage takes negligible time, the worst case running time of NDOP-Solve occurs when all n items are stacked upon one another. In this case, all $n!$ possible orderings must be examined for feasibility.

Unlike NDOP-Solve, QOP-Solve is not necessarily complete even if the offline planner is complete. This is because the offline planner may commit early to a bad choice because it assumes that it has control over future item ordering, as shown in Fig. 5. However, if it does return a solution, then this solution is feasible even if a heuristic offline planner is used.

2) DEPSET-COMPAT Is NP-Complete: We observe that Verify-CDG-Coverage terminates extremely quickly in many cases, but can exhibit exponential behavior. An example is shown in Fig. 6(a), in which each of the $m = n$ plans has one item depending on all other items. At each level ℓ of the recursion tree, there are $n-\ell$ valid CDGs, and all $n-\ell$ vertices are valid. Hence, the function is called $O(n!)$ times. In fact, we prove the following theorem.

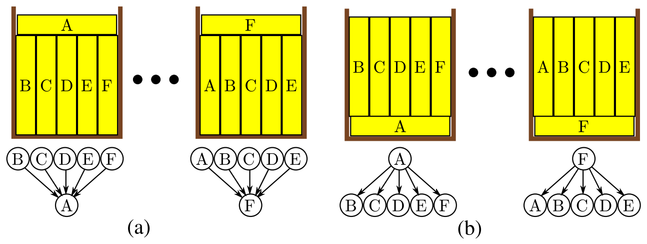

Fig. 6. (a) Worst case behavior of Verify-CDG-Coverage occurs in an instance with n plans, where $n-1$ "books" are stacked vertically with the nth book stacked horizontally on top. Every possible order of $k \le n$ items is compatible with a set of $n-k+1$ plans, and a recursion depth of n is required. (b) With a slightly different stacking, the dependence graphs are reversed. Only a depth 1 recursion is needed due to the singleton pruning step, and therefore, running time is polynomial. (a) Verification is $O(n!)$. (b) Verification is $O(n^2)$.

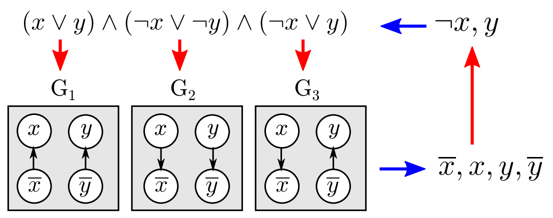
$(x \vee y) \wedge (\neg x \vee \neg y) \wedge (\neg x \vee y)$
$\neg x, y$
G_1
G_2
G_3
$\overline{x}, x, y, \overline{y}$

Fig. 7. Illustrating the reduction from SAT. Each clause (upper left) is converted into a dependence graph (lower left), and a counterexample (lower right) ordering corresponds to an SAT solution (upper right).

Theorem 1: DEPSET-COMPAT is NP-complete.

Proof: The proof is via polynomial time reduction from the classical NP-complete problem 3-SAT [21]. A 3-SAT instance consists of n Boolean variables $x_1, \ldots, x_n$ and a logical expression in disjunctive normal form with m clauses

$$(y_{11} \vee y_{12} \vee y_{13}) \wedge \cdots \wedge (y_{m1} \vee y_{m2} \vee y_{m3}) \tag{9}$$

where y_{ij} indicates either a variable or its negation, i.e., $y_{ij} = x_k$ or $y_{ij} = \neg x_k$. We transform any 3-SAT instance in this form into the complement of a DEPSET-COMPAT instance on $2n$ vertices and up to m dependence graphs. That is, when 3-SAT has a solution, the DEPSET-COMPAT version returns an incompatible ordering which corresponds to a 3-SAT solution, and when 3-SAT has no solution, the DEPSET-COMPAT version returns "all compatible." This construction is shown in Fig. 7 for a 2-SAT instance.

Specifically, let $\mathcal{I} = \{v_1, \ldots, v_n, \overline{v}_1, \ldots, \overline{v}_n\}$ be the vertex set. Consider the ith conjunctive clause in (9). First, if the same variable and its negation appear in the same clause (e.g., $x_4 \vee \neg x_4 \vee x_6$), we drop the clause because it is satisfied via any assignment. Otherwise, we construct a dependence E_i as follows. If $y_{ij} = x_k$ for some k, construct an edge $(\overline{v}_k, v_k)$. If $y_{ij} = \neg x_k$, construct an edge $(v_k, \overline{v}_k)$. In the first case, this means that if this dependence is violated, then v_k will appear before $\overline{v}_k$ in the ordering. In the second case, the reverse is true. This is repeated for each $j = 1, 2, 3$ and $i = 1, \ldots, m$.

If the DEPSET-COMPAT$(\mathcal{I}, E_1, \ldots, E_m)$ instance constructed in this way returns an incompatible ordering, we observe whether each v_k appears before $\overline{v}_k$. If so, we assign $x_k \leftarrow$True, and if not, $x_k \leftarrow$False. The variables $x_1, \ldots, x_k$ are then a solution to 3-SAT. This holds because in every clause $y_{i1} \vee y_{i2} \vee y_{i3}$, the dependence graph E_i is violated in such a way that makes the clause true.

Conversely, if the 3-SAT instance has a solution $(x_1, \ldots, x_n)$, the DEPSET-COMPAT instance has an incompatible ordering. It is constructed as follows: place v_k before $\overline{v_k}$ if $x_k = T$, and $\overline{v_k}$ before v_k if $x_k = F$. This ordering violates at least one constraint in each dependence graph.

Since each step in the reduction is polynomial time and NP-complete, DEPSET-COMPAT is NP-hard. It is also in NP since a nondeterministic recursion could enumerate all possible orderings and check their validity in $O(mn)$ time. ■

What is interesting about this reduction is that DEPSET-COMPAT is hard even if restricted to seemingly easy classes of dependence graphs, e.g., separable, bipartite graphs with at most three dependencies! Experimentally, we have observed that DEPSET-COMPAT problems corresponding to hard 3-SAT problems (e.g., with a clause-to-variable ratio of ∼4.24 [22]) also exhibit exponential complexity when solved via Algorithm 1.

VI. Planning Heuristics

Although DEPSET-COMPAT is NP-complete, the computation time of NDOP-Solve and QOP-Solve is dominated by time spent in the offline planner because each plan requires searching over 6-D object pose. The number of plans requested, and hence, the overall running time is greatly dependent on the number of orderings compatible with previous offline plans. Hence, it would be beneficial if the offline planner would generate packing plans that maximize compatibility. We employ some heuristics that speed up the approach in common scenarios.

A. Dependence Minimization Heuristic

Constructive packing chooses an item's location based on certain placement heuristics, such as deepest bottom-left-first [11] or heightmap minimization (HM) [11], [15], to maximize packing density. For nondeterministic packing, we would like to generate plans with a few dependencies. We introduce a *dependence count* (DC) heuristic that measures the number of items underneath the item at the given placement (i.e., number of ancestor nodes in the CDG). Our implementation uses a heuristic that is a weighted sum of HM and DC.

B. Matching Prior Placements

In QOP-Solve it is beneficial for the offline planner to place as many "free" items (i.e., those not in $\sigma^{\text{fixed}}_{1:j}$ or $\sigma^{\text{next}}_{j+1:k}$) as possible in the same location as the prior plan, since this will maximize the likelihood that the plan is compatible with other branches in the search tree. To implement this heuristic, when packing a free item, the location in P^{prior} is checked for feasibility before any other locations are tested.

C. Container Optimization Heuristics

During container optimization, it is helpful to limit exponential growth in running time by replacing the infinite loop in

Line 3 of NDOP-Solve (Algorithm 2) with a fixed number of iterations, or break the recursion of QOP-Solve (Algorithm 3) after the policy graph has grown too large. As the containers grow wider/longer, the number of dependencies decreases because all items will be packable in fewer layers. With a large enough container, all items are packable in a single layer. Hence, if there exists a sufficiently large container, the optimization version will always terminate with a feasible, but a possibly suboptimal solution.

VII. Packing With Equivalence Classes

A straightforward extension of the NDOP and QOP problems is to handle equivalent items in the packing process. This is a common case for larger itemsets. Because of items in the same equivalence class are replaceable, it reduces the number of possible orders that need to be considered by the nondeterministic packer. For example, if a consumer orders five item As and three item Bs, then the set of possible orderings is the set of unique strings containing five As and three Bs. This gives only 56 possible orders compared with $8! = 40\,320$ in the case where all 8 objects are distinct.

To address this case, an equivalence relation $\sim$ is defined over the itemset $\mathcal{I}$, and the quotient space $\tilde{\mathcal{I}} = \mathcal{I}/\sim$ is the set of unique items. The notion of CDG compatibility must be extended to allow for equivalence classes. We use a recursive definition. Let $G = (\mathcal{I}, E)$ be the CDG of a feasible plan P. An ordering $\sigma'_{1:n}$ is compatible with G if:

1) $n = 0$ or;
2) σ'_1 is *equivalent to* a root in G AND $\sigma'_{2:n}$ is compatible with G', where G' is G with σ'_1 removed.

Determining whether $\sigma'_{1:n}$ is compatible with a plan is somewhat more expensive than the standard $O(|G|)$ procedure. Any arriving item can be matched with any root in the same equivalence class; therefore, determining compatibility requires maintaining multiple hypothetical matches (and hence, multiple reduced CDGs G'). We verify compatibility in depth-first fashion, speeding up search using the all-root base case and treating all singletons as equivalent.

A. NDOP With Equivalence

Using this definition, we can extend Verify-CDG-Coverage to handle equivalence classes as listed in Algorithm 5. The algorithm only has minor modifications to handle equivalence classes when testing for roots and singletons, and rather than looping over items it loops over classes. Simply using Verify-CDG-Coverage-Eq in NDOP-Solve (Algorithm 2) will address the problem of NDOP planning with equivalence classes.

B. QOP With Equivalence

Algorithm 6 presents a solver for QOP with equivalence classes that modify QOP-Recurse (Algorithm 3) in a similar fashion to Verify-CDG-Coverage-Eq. The main difference, in this case, is that we maintain the item sequence while building the policy tree, but the branching should be understood as no longer performed on items but rather equivalence classes. Moreover, whereas in the prior QOP-Recurse algorithm, each plan $P_i \in \mathcal{P}_N$ is associated with a single-dependence subgraph associated with the removal of packed items, in this case, each plan has potentially multiple compatible subgraphs. These subgraphs can be propagated through the recursion along with N in a manner similar to lines 8–13 of Verify-CDG-Coverage-Eq (Algorithm 5).

Algorithm 5 Verify-CDG-Coverage-Eq($\mathcal{I}, \mathcal{G}$)

```
input : a set of items $\mathcal{I}$
        list of dependency graphs $\mathcal{G} = (G_1, \ldots, G_m)$
1  if there exists $v \in \mathcal{I}$ that is not equivalent to a root in any graph $G_i$ then return "$v$ incompatible";
2  if any $G_i \in \mathcal{G}$ has no edges then return "all compatible";
3  while $\exists c \in \tilde{\mathcal{I}}$ such that there is a singleton of class $c$ in every graph $G_i \in \mathcal{G}$ do
4      Remove an item of class $c$ from $\mathcal{I}$;
5      Remove an singleton of class $c$ from each $G_i \in \mathcal{G}$;
6  end
7  for $c \in \tilde{\mathcal{I}}$ do
8      $\mathcal{G}^c \leftarrow ()$;
9      for $i = 1, \ldots, m$ do
10         for roots $v$ in $G_i$ with class $c$ do
11             Append $G_i$ to $\mathcal{G}^c$, but with $v$ removed;
12         end
13     end
14     Let $v$ be an item in $\mathcal{I}$ with class $c$;
15     $r \leftarrow$Verify-CDG-Coverage-Eq($\mathcal{I} \setminus \{v\}, \mathcal{G}^c$);
16     if $r =$"$\sigma_{1:j}$ incompatible" then
17         return "$v, \sigma_{1:j}$ incompatible"
18 end
19 return "all compatible"
```

VIII. Buffered Nondeterministic Planning

Lastly, we present *k-buffered* NDOP and QOP variants, which are hybrids of NDOP/QOP and the buffered online packing problem, in which the packer has a "buffer" that can hold up to k items before having to make a container choice [23], [24]. This formulation spans a continuum between offline packing ($k \geq n - 1$) and nondeterministic planning ($k = 0$). It is also fairly realistic for packing setups in warehouses since it is typically possible to augment the container with a temporary space where items may be set aside. Moreover, multi-armed packing robots (or humans) can temporarily "store" items in their hands, where k is the number of hands minus 1. The overall benefit of this approach is that the number of plans that must be computed by the offline planner decreases with k. Specifically, if $N(k)$ is the number of offline plans computed, $N(k)$ is nonincreasing with k, and if the buffer is large enough, the problem becomes equivalent to standard offline planning [$N(n-1) = 1$].

The other potential benefit of a buffered approach is that some infeasible NDOP/QOP problems can become feasible because a buffer provides more control over the packing order so that items might be packed more densely. Fig. 8 gives an example of such a problem.

Algorithm 6 QOP-Recurse-Eq(N)

```
1  Let $\mathcal{I}$ be the set of items not in $\sigma_{1:\ell}$ ;
2  if for any $P \in \mathcal{P}_N$ the subgraph of $P$ induced by $\mathcal{I}$ has no edges then return "success";
3  for all classes $c \in \tilde{\mathcal{I}}$ do
4      Let $\sigma_{\ell+1}$ be an item in $\mathcal{I}$ with class $c$;
5      if any plan in $\mathcal{P}_N$ is compatible with an item of class $c$ then
6          Let $T_c$ be the location compatible with the most plans in $\mathcal{P}_N$;
7          $\mathcal{P}_C \leftarrow \{P' \in \mathcal{P}_N | \quad P'$ is compatible with $T_c\}$;
8      else
9          Let $P$ be any plan in $\mathcal{P}_N$, or nil if $\mathcal{P}_N = \emptyset$;
10         $P' \leftarrow$Offline-Pack($\sigma_{1:\ell}, P, \sigma_{\ell+1}$);
11         if $P' =$"failure" then return "failure";
12         $\mathcal{P}_C \leftarrow \{P'\}$;
13         For all ancestors $A$ of $N$, add $P'$ to $\mathcal{P}_A$;
14     end
15     $C \leftarrow$ add-child($N, \sigma_{\ell+1}, \mathcal{P}_C$);
16     if QOP-Recurse-Eq($C$) fails then return "failure";
17 end
18 return "success"
```

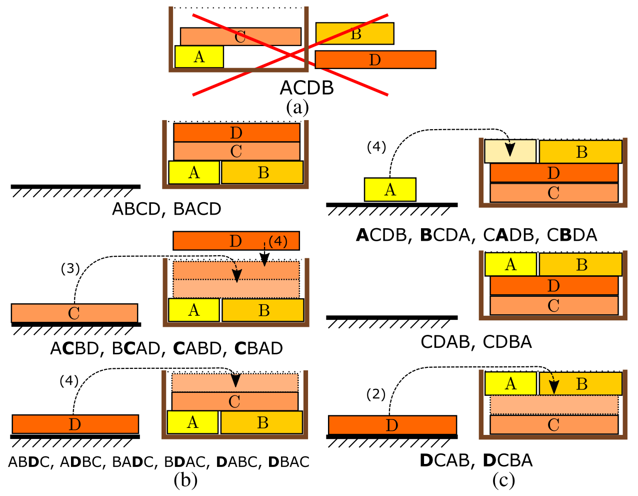

Fig. 8. (a) Problem that has no standard NDOP solution, but is feasible with a buffer of size $k = 1$. Three plans are sufficient to cover all 4! orderings. (b) Plan 1 covers 12 orderings: 2 without using the buffer, 4 placing C in the buffer, and 6 placing D in the buffer. Bold letters indicate the buffered item, and numbers indicate the packing order. (c) Plan 2 covers 8 orderings. Plan 3 (not shown) swaps C and D to cover the remaining 4 orderings. (a) No offline plan for ACDB. (b) Plan 1. (c) Plan 2.

A. Buffered NDOP

In order to compute and use k-buffered NDOP policies, we must cope with the fact that the arrival and packing order are not necessarily the same; an arrived item not immediately compatible with a plan can be set aside for later packing. Three changes are made to NDOP-Solve. First, the notion of compatibility must be revised to manage the buffer and decide when to put an item aside. Second, the NDOP search must be augmented to maintain buffer states. Third, we cannot generate a single-counterexample ordering for offline planning because there may be a feasible ordering that uses the buffer in a different manner than expected.

We start by defining the notion of *buffer compatibility*. Given a plan, its CDG, and a new order, there is a straightforward greedy policy for packing while maintaining a minimal buffer. There are two rules to follow for a given item v and buffer B.

1) If all predecessors of v in the CDG are packed, then pack v. Otherwise, put v in the buffer.
2) If, for any buffered item $w \in B$, all predecessors of w in the CDG are packed, then pack w (removing it from the buffer).

If the number of items held in the buffer at any time during this process is less than or equal to k, then there exists a feasible packing order for this plan. This can be checked in $O(nk)$ time by incrementally removing root nodes from the CDG when the corresponding item is packed since Step 1 is $O(1)$ time and Step 2 is $O(k)$ time. Let us call Step 2 *cleaning the buffer*. Note that cleaning the buffer must apply the check in Step 2 repeatedly if there are multiple items in the buffer that are unblocked by the packing of a new item. For example, if the CDG is $v_3 \rightarrow v_2 \rightarrow v_1$ and the packing order is v_1, v_2, and v_3, then the buffer is $\{v_1$ and $v_2\}$ when item v_3 is packed, and packing v_3 unblocks v_2 then v_1 in turn.

We use this reasoning in a buffered Verify-CDG-Coverage algorithm, given in Algorithm 7. Given a set of plans $P_1, \ldots, P_m$ with CDGs $G_1, \ldots, G_m$, the new subroutine explores partial orderings of items while maintaining the minimal buffer $B_1, \ldots, B_m$ for each plan. The buffers are initialized to empty sets. The base cases are modified as follows.

1) There exists a vertex v that is not a root in any G_i and all $|B_j| = k$. Then, any order that starts with v is a counterexample.
2) The number of root nodes in some G_i is at least $n - k + |B_i|$. The policy is feasible because P_i is buffer compatible with all orderings.

Base case 1 is less aggressive at declaring failure than in the nonbuffered case, because even if a vertex is nonroot, then it can still be added to the buffer while there is room. Base case 2 is more aggressive at declaring success because we can terminate recursion even when a CDG is nonempty. If the number of nonroot nodes is less than or equal to k, then the buffer can hold them all regardless of the presentation order.

The recursive step is modified so that, first, the buffer for each plan is cleaned. Then, upon visiting a vertex v, a plan P_i and its buffer B_i are retained if either v can be packed in P_i or there is room in the buffer ($|B_i| < k$). If v can be packed directly, there is no sense in considering the option of placing it in the buffer, since the greedy policy is optimal.

Simply replacing Verify-CDG-Coverage with Verify-CDG-Coverage-Buf in NDOP-Solve (Algorithm 2) will correctly solve buffered problems in which a nonbuffered NDOP solution exists. But for infeasible NDOP problems, e.g., the one in Fig. 8, this approach may fail to find a buffered solution. Specifically, the offline-pack call in line 7 of NDOP-Solve may fail on the counterexample $\sigma_{1:\ell}$ even though some offline plan

Algorithm 7 Verify-CDG-Coverage-Buf($\mathcal{I}, \mathcal{G}, \mathcal{B}, k$)

```
input : a set of items $\mathcal{I}$
        dependency graphs $\mathcal{G} = (G_1, \ldots, G_m)$
        buffers $\mathcal{B} = (B_1, \ldots, B_m)$
        buffer size $k$
1  Clear buffers $B_i$ in their respective graphs $G_i$;
2  if there exists $v \in \mathcal{I}$ for which $v$ is not a root in any $G_i$
   and all $|B_i| = k$ then return "$v$ incompatible";
3  if there exists $G_i \in \mathcal{G}$ with at least $n - k + |B_i|$ root
   nodes then return "all compatible";
4  Remove all vertices $v$ from $\mathcal{I}$, graphs, and buffers that
   are singletons in every $G_i$, $i = 1, \ldots, m$;
5  for $v \in \mathcal{I}$ do
6      $\mathcal{G}^v \leftarrow ()$, $\mathcal{B}^v \leftarrow ()$;
7      for $i = 1, \ldots, m$ do
8          if $v$ is a root in $G_i$ then
9              Append $G_i$ to $\mathcal{G}^v$, but with $v$ removed;
10             Append $B_i$ to $\mathcal{B}^v$;
11         else if $|B_i| < k$ then
12             Append $G_i$ to $\mathcal{G}^v$;
13             Append $B_i \cup \{v\}$ to $\mathcal{B}^v$;
14     end
15     $r \leftarrow$Verify-CDG-Coverage-Buf($\mathcal{I} \setminus \{v\}, \mathcal{G}^v, \mathcal{B}^v$);
16     if $r =$"$\sigma_{1:j}$ incompatible" then
17         return "$v, \sigma_{1:j}$ incompatible"
18 end
19 return "all compatible"
```

maybe is buffer compatible with this order (e.g., the counterexample ACDB to plan 1 in Fig. 8). To ensure completeness, the offline planner must be made buffer-aware. Specifically, if there is an offline plan that *packs, buffers, or debuffers* items in the same order as they appear in $\sigma_{1:\ell}$, it is a valid solution to the requested counterexample. Note that the plan must describe how all n items must ultimately be packed, but the planner now has more freedom to choose which items are packed first. Let us call this modified oracle *Offline-Pack-Buf*.

If the offline planner uses a heuristic item ordering strategy, as it does in our implementation, the best way to implement Offline-Pack-Buf is to allow it to delay packing up to k items if subsequent items have higher priority, or an item fails to be placed. (It is an open question about how to accomplish this efficiently for a complete offline planner; a correct but slow approach would enumerate all subsequences of $\sigma_{1:\ell}$ with up to k buffered objects, and try finding an offline plan for each of them.)

B. Buffered QOP

For QOP, each node in the policy tree may buffer the item, rather than pack it in a given location, delaying the decision of the packing location to some descendant node. This requires each QOP node N maintain a *packing list* of items and their locations $(\sigma_{N1}, T_{N1}), \ldots, (\sigma_{Nj}, T_{Nj})$ packed upon the arrival of an item. An empty list ($j = 0$) indicates the arrived item is buffered.

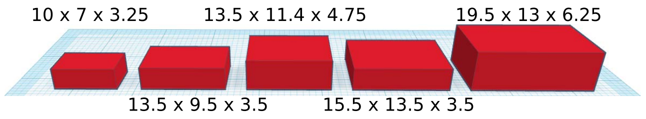

Fig. 9. Dimensions (in inches) of containers 1–5 used in our experiments, in order of increasing length + girth.

To build intuition, we describe how a naïve k-buffered QOP variant can be implemented by modification of the naïve QOP algorithm of Section V-E. Each node is associated with an arrival order $\sigma_{1:\ell}$, a packing list L, a feasible plan $P = (\sigma_{1:n}, T_{1:n})$, and an optimal buffer B at step ℓ. We also define the dependence subgraph G of the CDG of P when the items in $\sigma_{1:\ell} \setminus B$ are removed. In each recursive call, perform the following steps.

1) For each item $\sigma_{\ell+1}$ not in $\sigma_{1:\ell}$, repeat Steps 2–4.
2) Call Offline-Pack-Buf, replacing $\sigma_{1:\ell}$ in (8) with $\sigma_{1:\ell} \backslash B$ and using B as the initial buffer. If this fails, return failure. (Note that this may choose to buffer $\sigma_{\ell+1}$.)
3) Create a child node whose pack list is initialized with the pack or buffer decision for $\sigma_{\ell+1}$. Initialize its dependence subgraph G and buffer B.
4) Continue recursively on the child.

If a node is a leaf, then the node's buffers should be cleared. Note that unlike NDOP, QOP actually gains power by delaying packing decisions until the buffer is full. Hence, there is no need to pack until $|B| = k$. Delaying also maximizes our ability to reuse offline plans across many policy nodes.

A more refined QOP algorithm reuses offline plans, such as Algorithm 3. We modify each QOP search node N to contain elements.

1) The arrival sequence $\sigma_{1:\ell}$ leading to and including N.
2) The packing list $L_N = (\sigma_{N1}, T_{N1}), \ldots, (\sigma_{Nj}, T_{Nj})$ of decisions made upon the arrival of item σ_ℓ.
3) The buffer B_N associated with the packing sequence leading to and including N.
4) A list of plans $\mathcal{P}_N = (P_1, \ldots, P_m)$ that are buffer compatible with $\sigma_{1:\ell}$.
5) A list of dependence subgraphs $\mathcal{G}_N = (G_1, \ldots, G_m)$ corresponding to the plans in $\mathcal{P}_N$, restricted to the complement of $(\sigma_{1:\ell} \setminus B_N)$.

The resulting algorithm is listed in Algorithm 8.

IX. Experiments

Our experiments test the NDOP and QOP algorithms with random item sets of 3-D scanned objects from the Amazon Picking Challenge (APC) 2015 [25] and Yale-CMU-Berkeley (YCB) [26] data sets (94 objects total). The containers used in these experiments are the five boxes used in the Amazon Robotics Challenge 2017 (Fig. 9) and are sorted by the length + girth metric (length + 2×width + 2×height), a commonly used shipping measurement. All experiments were performed on an Amazon EC2 m5d.12× large instance.

Our experiments use three testing data sets in which the item sets have different size: 1) typical shopping carts of 2–5 items; 2) large sets of five items; and 3) stress tests with ten items. For each category, we generate 1000 random item sets by drawing

Algorithm 8 QOP-Recurse-Buf(N, k)

```
input : policy tree node $N$ at level $\ell$
        buffer size $k$
1  Compute dependency subgraphs $\mathcal{G}_N$;
2  if any $G_i \in \mathcal{G}_N$ would have at least $n - k + |B_N|$ root nodes after cleaning $B_N$ then return "success";
3  for all items $\sigma_{\ell+1} \notin \sigma_{1:\ell}$ do
4      if $|B_N| < k$ then
5          $L_C \leftarrow ()$;
6          $B_C \leftarrow B_N \cup \{\sigma_{\ell+1}\}$;
7          $\mathcal{P}_C \leftarrow \mathcal{P}_N$;
8      else if any item in $B_N \cup \{\sigma_{\ell+1}\}$ is a root of any $G_i$ then
9          Let $\sigma \in B_N \cup \{\sigma_{\ell+1}\}$ and $T_\sigma$ be the item and location compatible with the most plans in $\mathcal{P}_N$;
10         $L_C \leftarrow (\sigma, T_\sigma)$;
11         $B_C \leftarrow B_N$;
12         $\mathcal{P}_C \leftarrow \{P' \in \mathcal{P}_N \mid P'$ is compatible with $\sigma$ and $T_\sigma\}$;
13     else
14         $P \leftarrow$ any plan in $\mathcal{P}_N$, or nil if it is empty;
15         $P' \leftarrow$Offline-Pack-Buf($\sigma_{1:\ell} \setminus B_N, P, \sigma_{\ell+1}, B_N$);
16         if $P'$ ="failure" then return "failure";
17         $L_C \leftarrow$ the next packed item in $P'$;
18         $B_C \leftarrow$ the resulting buffer;
19         $\mathcal{P}_C \leftarrow \{P'\}$ ;
20         For all ancestors $A$ of $N$, add $P'$ to $\mathcal{P}_A$;
21     end
22     $C \leftarrow$ add-child($N, \sigma_{\ell+1}, L_C, \mathcal{P}_C, B_C$);
23     if QOP-Recurse-Buf($C, k$) fails then return "failure";
24 end
25 return "success"
```

TABLE I
NDOP CONTAINER OPTIMIZATION RESULTS

Items	Success (%)	Time (mean / max, s)	# planner calls (mean / max)
2–5	99.3	73.0 / 1,813	1.3 / 9
5	96.9	94.4 / 1,417	1.6 / 14
10	64.0	1,048 / 33,300	5.2 / 118

items at random and verifying with an offline planner that there exists is a feasible packing in one of the five containers. In the 2–5 categories, 250 item sets of each size are included.

A. NDOP Results

The results for the NDOP planner, running in container optimization mode, are summarized in Table I. Fig. 10 shows a solution. As might be expected, running time and the number of offline planner calls increase with the number of items, but we do not observe the exponential running time of worst case instances. Other experiments suggest the dependence minimization heuristic reduces mean and maximum running times by approximately 20% and 50%, respectively.

Observe that the success rate also drops with increasing numbers of items, as the ten-item offline plans tend to be tightly packed even in the largest container. Note that it is not known whether an NDOP solution exists in these instances; therefore, we cannot determine whether the solver is failing incorrectly. Fig. 11 shows the distribution of the minimum-cost container found. In cases with five or fewer items, NDOP-Solve often successfully packs in the same box as the offline planner. Observe that with five items, approximately 5% of test cases require container 5, even though they can be packed offline in containers 1–4. Fig. 12 shows an example of such a case.

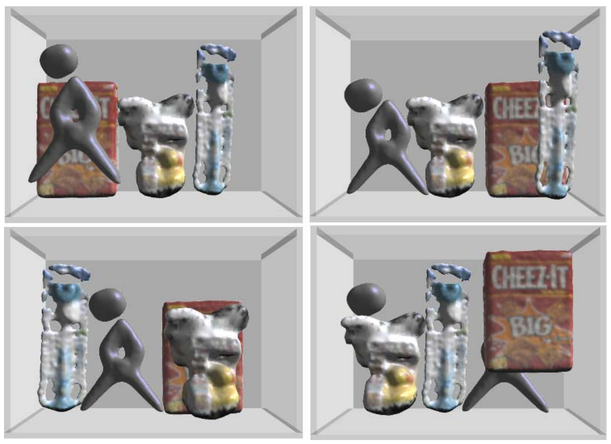

Fig. 10. NDOP solution for packing five items in container 5.

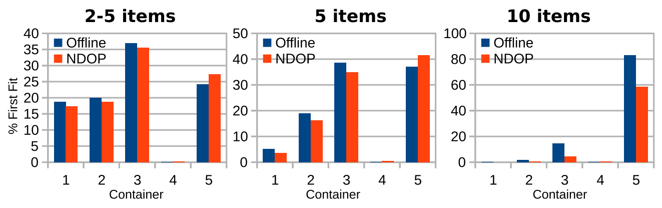

Fig. 11. Distribution of the solution container for offline/NDOP container optimization and various itemset sizes. Most small instances can be packed in any order, but with more items the variability in order requires larger boxes.

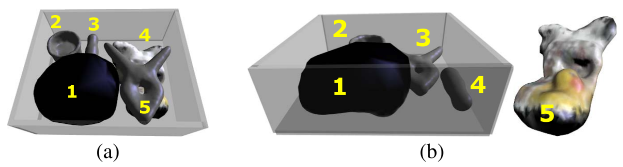

Fig. 12. Failure case for packing five items in container 3. The offline planner solves for the arrival order in (a) but fails on the order in (b) because there is insufficient remaining space for the fifth item.

B. QOP Results

Performance results for QOP-Solve in container 5 are given in Table II, with a representative solution shown in Fig. 13. Up to five items, the success rates are quite similar to NDOP-Solve, but the maximum running times and the number of offline planner calls tend to be significantly larger. Other experiments suggest that employing the matching heuristic improves average and maximum running time by over 50%, which explains the surprising result that QOP-Solve is faster

TABLE II
QOP PLANNING RESULTS IN CONTAINER 5

Items	Success (%)	Time (mean / max, s)	# planner calls (mean / max)
2–5	99.4	22.4 / 1,520	1.4 / 43
5	97.0	65.1 / 5,800	2.1 / 46
10	43.7	2,850 / 85,478	45.8 / 5,363

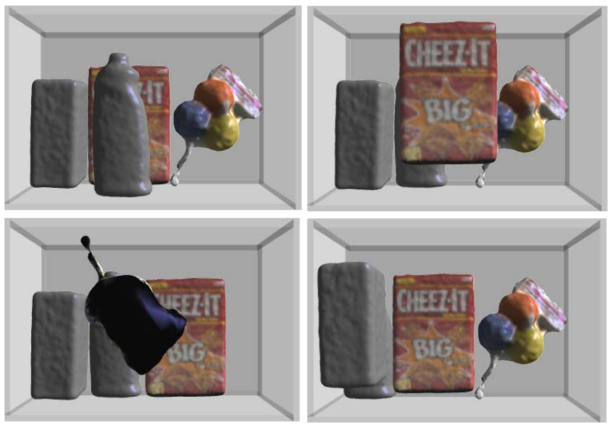

Fig. 13. QOP solution for four items in container 5. Due to the matching heuristic, only four plans are needed (one for each item placed last).

than NDOP-Solve on average. QOP-Solve struggles with 10 items, with a long-tailed distribution: 24 instances could not be solved within a 24-h cutoff.

C. Results for Equivalence and Buffered Variants

Fig. 14 shows how performance changes when the buffered and equivalence class extensions to the NDOP planner are introduced. This experiment uses an artificial model of an offline planner, where the oracle always returns a valid plan, and its CDG is a random subset of the maximally dependent CDG, where each item is dependent on all prior items. (Offline planning time is negligible in this model.) Fig. 14(a) is the worst case where each CDG is maximally dependent, so each plan is only compatible with a single ordering and hence the standard NDOP problem must generate $n!$ oracle calls. In Fig. 14(b), each edge of the maximally dependent CDG is retained with probability 0.5. The total verification time budget is capped at 10 min, and therefore, the standard NDOP-Solve cannot solve Fig. 14(a) for $n \geq 7$ and Fig. 14(b) for $n \geq 9$. "Equiv k" indicates that items are allocated among k equivalence classes. Each class is balanced to hold approximately the same number of items. For both dependence models, larger buffer sizes and fewer equivalence classes lead to more scalable performance.

Figs. 15 and 16 demonstrate these variants on random itemsets drawn from the APC/YCB data sets. To test the effects of equivalence classes, we generated 1000 new 10-item itemsets, where only 2 item classes were chosen at random, and the split between classes was chosen between 4–6 items at random. For each itemset, it was verified that with offline planning, all items fit in at least one of the five Amazon

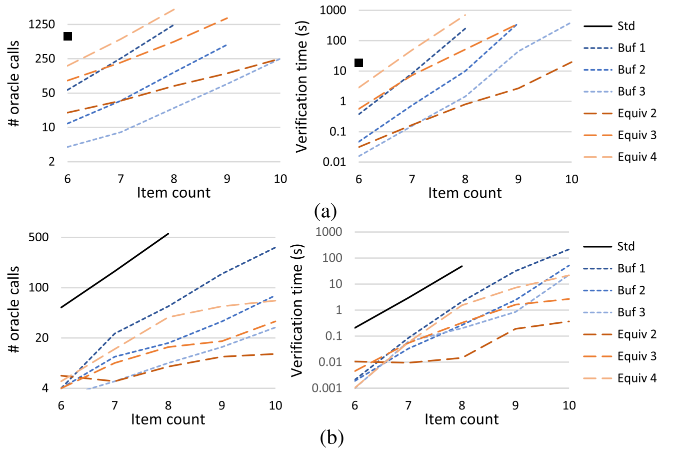

Fig. 14. Comparing performance of standard NDOP-Solve (Std) against the k-buffered (Buf k) and equivalence class (Equiv k) variants described in Sections VIII-A and VII-A, respectively, on artificial examples. Number of offline plans and coverage verification time are plotted for varying item counts (note the logarithmic scale). (a) Worst case performance, with each plan inducing a fully dependent CDG. Note that Std cannot handle more than six items within 10-min running time. (b) Performance when each plan contains 50% of dependencies, chosen at random. (a) 100% dependence model. (b) 50% dependence model.

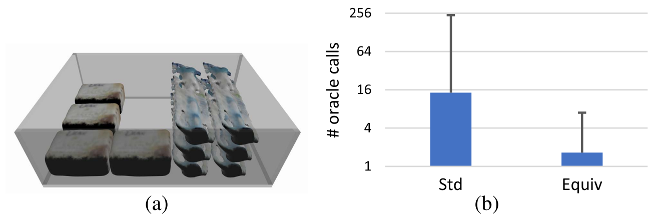

Fig. 15. Equivalence-class NDOP on items from the APC/YCB data sets. Two item classes are selected at random, with 4–6 items per class. (a) Planner covers all orderings with a single offline plan. (b) Oracle call counts for standard NDOP-Solve (Std) and equivalence-class NDOP-Solve (Equiv), over 1000 itemsets. Column indicates average, and error bar indicates maximum. (Note logarithmic scale.)

containers. Fig. 15(a) demonstrates that all 10! orders of the given items can be covered in a single plan. In Fig. 15(b), we compare standard NDOP-Solve to the equivalence-class aware variant. With equivalence class information, the average number of offline planner calls is 1.65 on average, and 7 at most. Without, the average and maximum are 14.45 and 239, respectively. Next, we evaluate buffering. Fig. 16(a) shows a representative output of the buffered planner. For this example, standard NDOP-Solve finds a covering with 13 offline plans, but 1-buffered NDOP finds a covering with the 2 offline plans shown. The statistics in Fig. 16(b) show that with five-item packing, the number of oracle calls drops very close to 1 as the buffer size increases. A similar pattern is observed in ten-item packing [Fig. 16(c)]. Moreover, the success rate increases dramatically as the buffer size increases, because the larger buffer gives the planner more opportunity to solve badly ordered arrival sequences in boxes with high packing density.

D. Discussion

These experiments illustrate several interesting characteristics of nondeterministic packing problems and our solvers,

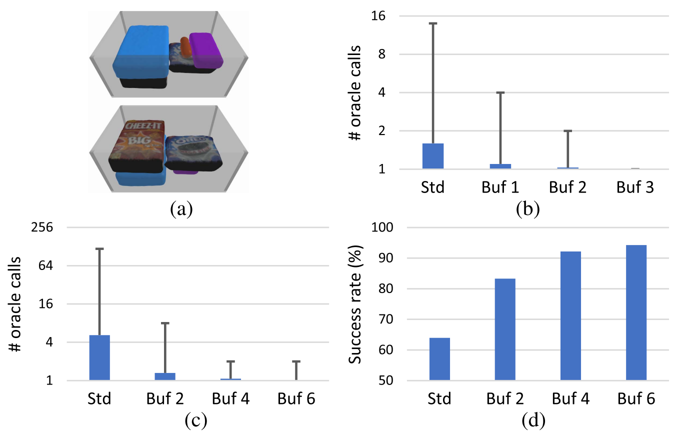

Fig. 16. Buffered NDOP on items drawn from the APC/YCB data sets. (a) With a buffer size of 1, a five-item solution can be covered with two offline plans. (b)–(d) Statistics over 1000 itemsets compare standard NDOP (Std) with k-buffered variance (Buf k). Columns indicate average, and error bars indicate maximum. (Note: oracle call counts plotted on logarithmic scale.) (a) Five-item, one-buffered plan. (b) Five-item oracle calls. (c) Ten-item oracle calls. (d) Ten-item success rate.

which are summarized here. First, handling nondeterminism may require sacrificing packing density (i.e., the use of larger containers). Moreover, the gap between the optimal packing density and the worst case density increases as the number of items grows. This is because the worst case ordering creates gaps in the container that cannot be filled by later items.

Second, our solvers can reach intractable running times with more than six or seven items. Although most orders tend to be small, it is worthwhile to consider how to address this problem. One option is parallelization, where several orderings are considered in parallel with cloud computing resources. In addition, since equivalence classes help significantly in cutting down computation time, it would be interesting to examine how to plan with approximately equivalent items in such a way that small differences between items in the same class do not invalidate the plan. In any case, a simple trick can be used to tradeoff between computation and packing density: simply optimize containers from large to small. Since NDOP and QOP can be solved trivially in large containers with stack-free packing, container optimization can be given the desired time budget while ensuring the plan is feasible.

Third, even small buffers can achieve tighter packing and one or more orders of magnitude improvements in computation time; therefore, adding buffer space to packing workcells can save in shipping costs and computational power. Large buffers, however, come at the cost of reduced utilization of warehouse space. Our algorithms could be used as the basis for analyzing these tradeoffs, using a large-scale simulation of items and orders that follow assumed statistical distributions.

X. Conclusion

This article formulated two novel packing problems with nondeterministic item ordering and presented practical solvers that handle irregular 3-D shapes and item sets up to size 10. It also presents extensions to the settings of equivalence classes and a buffer of size k, both of which reduce the number of offline plans needed (at a modest expense of making verification harder). This work opens up several interesting theoretical and practical questions, such as the minimal number of plans needed to guarantee NDOP coverage, whether complete QOP algorithms exist for rectilinear items, and whether restrictions on item shape can overcome exponential worst case complexity.

Additional problem variants would be interesting to study in future research. A nondeterministic formulation only addresses worst case order, but in the case where additional containers can be chosen to contain overflow items, it may be more appropriate to consider probabilistic formations and expected cost, particularly if there is probabilistic knowledge about the item ordering. Similarly, a planner could leverage knowledge about possible orderings, such as knowing that all items in an equivalence class arrive at once, e.g., on a pallet. In addition, we have not explored the possible use of *unpacking/repacking* to solve the QOP problem, in which the top layer(s) of objects acts as an additional dynamic buffer to give the packer greater flexibility.

References

[1] N. Correll *et al.*, "Analysis and observations from the first amazon picking challenge," *IEEE Trans. Autom. Sci. Eng.*, vol. 15, no. 1, pp. 172–188, Jan. 2018.
[2] S. Martello, D. Pisinger, and D. Vigo, "The three-dimensional bin packing problem," *Oper. Res.*, vol. 48, no. 2, pp. 256–267, Apr. 2000.
[3] S. S. Seiden, "On the online bin packing problem," *J. ACM (JACM)*, vol. 49, no. 5, pp. 640–671, Sep. 2002.
[4] F. Wang and K. Hauser, "Robot packing with known items and nondeterministic arrival order," in *Proc. Robot., Sci. Syst.*, 2019.
[5] S. Martello and D. Vigo, "Exact solution of the two-dimensional finite bin packing problem," *Manage. Sci.*, vol. 44, no. 3, pp. 388–399, Mar. 1998.
[6] B. S. Baker, E. G. Coffman, Jr., and R. L. Rivest, "Orthogonal packings in two dimensions," *SIAM J. Comput.*, vol. 9, no. 4, pp. 846–855, Nov. 1980.
[7] D. S. Johnson, A. Demers, J. D. Ullman, M. R. Garey, and R. L. Graham, "Worst-case performance bounds for simple one-dimensional packing algorithms," *SIAM J. Comput.*, vol. 3, no. 4, pp. 299–325, Dec. 1974.
[8] J. Egeblad, B. K. Nielsen, and A. Odgaard, "Fast neighborhood search for two- and three-dimensional nesting problems," *Eur. J. Oper. Res.*, vol. 183, no. 3, pp. 1249–1266, Dec. 2007.
[9] T. Kämpke, "Simulated annealing: Use of a new tool in bin packing," *Ann. Oper. Res.*, vol. 16, no. 1, pp. 327–332, Dec. 1988.
[10] X. Liu, J.-M. Liu, A.-X. Cao, and Z.-L. Yao, "HAPE3D—A new constructive algorithm for the 3D irregular packing problem," *Frontiers Inf. Technol. Electron. Eng.*, vol. 16, no. 5, pp. 380–390, May 2015.
[11] L. Wang, S. Guo, S. Chen, W. Zhu, and A. Lim, "Two natural heuristics for 3d packing with practical loading constraints," in *PRICAI: Trends in Artificial Intelligence*, B.-T. Zhang and M. A. Orgun, Eds. Berlin, Germany: Springer, 2010, pp. 256–267.
[12] M. Schwarz *et al.*, "Fast object learning and dual-arm coordination for cluttered stowing, picking, and packing," in *Proc. IEEE Int. Conf. Robot. Autom. (ICRA)*, May 2018, pp. 3347–3354.
[13] R. Shome *et al.*, "Towards robust product packing with a minimalistic end-effector," in *Proc. Int. Conf. Robot. Autom. (ICRA)*, May 2019, pp. 9007–9013.
[14] E. den Boef, J. Korst, S. Martello, D. Pisinger, and D. Vigo, "Erratum to 'the three-dimensional bin packing problem': Robot-packable and orthogonal variants of packing problems," *Oper. Res.*, vol. 53, no. 4, pp. 735–736, Aug. 2005.

[15] F. Wang and K. Hauser, "Stable bin packing of non-convex 3D objects with a robot manipulator," in *Proc. Int. Conf. Robot. Autom. (ICRA)*, May 2019, pp. 8698–8704.

[16] W. Wan, K. Harada, and K. Nagata, "Assembly sequence planning for motion planning," *Assem. Autom.*, vol. 38, no. 2, pp. 195–206, Apr. 2018.

[17] J. Egeblad, "Placement of two- and three-dimensional irregular shapes for inertia moment and balance," *Int. Trans. Oper. Res.*, vol. 16, no. 6, pp. 789–807, Nov. 2009.

[18] X. Han, F. Y. Chin, H.-F. Ting, G. Zhang, and Y. Zhang, "A new upper bound 2.5545 on 2D online bin packing," *ACM Trans. Algorithms (TALG)*, vol. 7, no. 4, p. 50, 2011.

[19] J. C. Trinkle, J.-S. Pang, S. Sudarsky, and G. Lo, "On dynamic Multi-Rigid-Body contact problems with Coulomb friction," *ZAMM - J. Appl. Math. Mech./Zeitschrift Für Angew. Math. Mech.*, vol. 77, no. 4, pp. 267–279, 1997.

[20] W. Wan, H. Igawa, K. Harada, H. Onda, K. Nagata, and N. Yamanobe, "A regrasp planning component for object reorientation," *Auto. Robots*, vol. 43, no. 5, pp. 1101–1115, Jun. 2019.

[21] S. A. Cook, "The complexity of theorem-proving procedures," in *Proc. 3rd Annu. ACM Symp. Theory Comput.*, 1971, pp. 151–158.

[22] J. W. Freeman, "Hard random 3-SAT problems and the Davis-Putnam procedure," *Artif. Intell.*, vol. 81, nos. 1–2, pp. 183–198, Mar. 1996.

[23] J. Csirik and D. S. Johnson, "Bounded space on-line bin packing: Best is better than first," *Algorithmica*, vol. 31, no. 2, pp. 115–138, Oct. 2001.

[24] L. Epstein and E. Kleiman, "Resource augmented semi-online bounded space bin packing," *Discrete Appl. Math.*, vol. 157, no. 13, pp. 2785–2798, Jul. 2009.

[25] *Rutgers APC RGB-D Dataset*. Accessed: Sep. 19, 2020. [Online]. Available: http://pracsyslab.org/rutgers_apc_rgbd_dataset

[26] *YCB Benchmarks—Object and Model Set*. Accessed: Sep. 19, 2020. [Online]. Available: http://ycbbenchmarks.org