Robot Packing with Known Items and Nondeterministic Arrival Order

Fan Wang
Department of Electrical and Computer Engineering
Duke University, Durham, NC 27708, USA
Email: fan.wang2@duke.edu

Kris Hauser
Department of Electrical and Computer Engineering
Duke University, Durham, NC 27708, USA
Email: kris.hauser@duke.edu

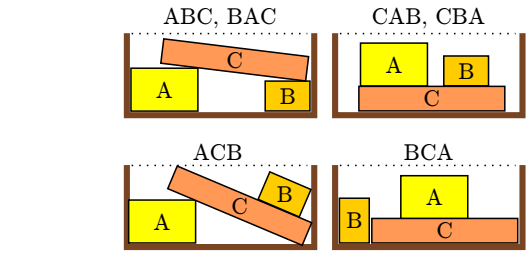
***Abstract*—This paper formulates two variants of packing problems in which the set of items is known but the arrival order is unknown. The goal is to certify that the items can be packed in a given container, and/or to optimize the size or cost of a container so that that the items are guaranteed to 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 item location as each subsequent item is revealed. Theoretical analysis demonstrates that even the simple subproblem of verifying feasibility of a packing policy is NP-complete. Despite this worst-case complexity, practical solvers for both NDOP and QOP are developed, and experiments demonstrate their application to packing irregular 3D shapes with manipulator loading constraints.**

I. Introduction

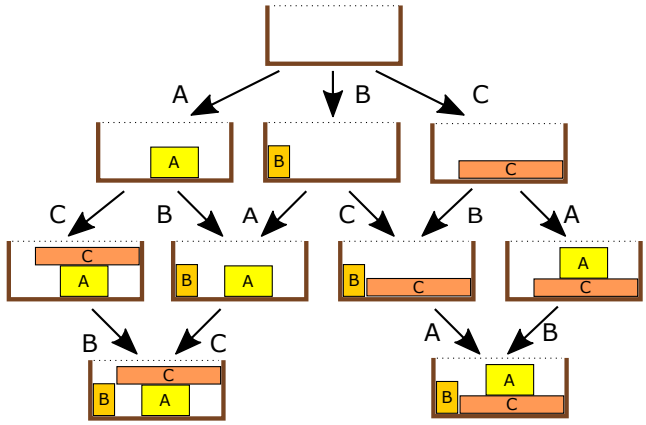
Interest in warehouse automation has grown rapidly with the growth of e-commerce and advances in robotics. Given the rapid progress in the field of robotic manipulation, the prospect of fully autonomous picking and packing robots is becoming increasingly likely in the near future [3], but relatively 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 [16]. In an *online* setting, the items are not known a priori and need to be placed as they arrive [18]. 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 paper 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 cases where item deliveries are human-controlled, 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 NDOP variant, the feasibility of the container is verified under all nondeterministic orders, but the arrival sequence is revealed



(a) Nondeterministically ordered packing (NDOP)



(b) Quasi-online packing (QOP)

Fig. 1. Feasible solutions for a 2D, 3-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.

before packing is executed. In the QOP variant, each object must be packed before the next item is revealed (Fig. 1).

We present a practical framework for solving NDOP and QOP problems that 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 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 reduction from 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 scalability of the approach, and experiments demonstrate that our approach can be realistically applied to irregular 3D shapes with item sets of size up to 10.

II. Related work

Packing algorithms have been studied extensively both for their theoretical interest and practical applications in shipping, manufacturing, and 3D printing. The vast majority of work considers rectilinear objects. State-of-the-art exact algorithms for the offline 2D and 3D bin packing problem use branch-and-bound approaches [15, 16]. Because exact methods have worst-case exponential complexity, heuristic methods and metaheuristic approaches have been developed, such as the Bottom-Left [1] and Best-Fit-Decreasing heuristics [12]. Heuristics are the only practical methods available for irregular shape packing (a.k.a. nesting [7]), since the freedom to rotate leads to a continuously infinite search space. Metaheuristic optimization methods [9, 13] simultaneously optimize the placements of all items, and constructive heuristics incrementally place items according to some scoring function [14, 20].

In the offline setting, the item set and packing order can be 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 like sheet metal cutting, but one should consider additional constraints when packing containers in practice. Prior work has enforced clearance of boxes along axis-aligned loading directions in 3D bin-packing by ensuring no previously-placed item lies along an extruded prism along at least one face of the box [5]. Recent work has also formulate collision-checking between the loading mechanism (human hand, forklift, robot hand, etc.) and already-placed items [19]. Stability constraints [6, 14, 19] also impose dependency 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.

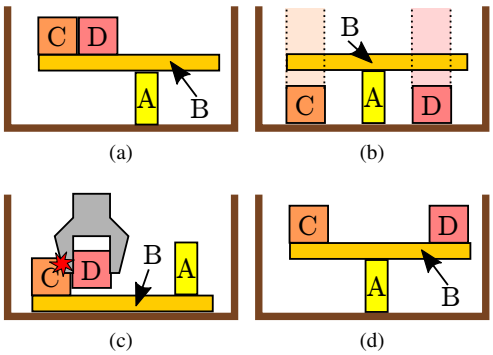


Fig. 2. Examples of plans that are infeasible for arrival order ABCD: (a) unstable, (b) items C and D collide with B along the loading direction, (c) and the path for the robot manipulator to grasp and load item D is infeasible. In (d), although ABCD is feasible, the prefix requirement is violated because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

In the online setting, an arbitrary item is presented to the algorithm, which then chooses a packing location [18]. Neither the item set nor the packing order are controllable, and often the item geometry can also be arbitrary as well. There are no guarantees that a given container can be packed, so 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 1D and 2D rectilinear bin packing settings [11, 18], but to our knowledge, no results are known for irregular shapes. In contrast, NDOP and QOP seek guaranteed packing in a single container when the item set is known, which is more appropriate for fulfillment applications. Our NDOP and QOP solvers can handle irregular shapes, and we prove that when they successfully return a policy, the answer is correct.

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

An 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 the below section.

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 i'th item to be the ordinal index in which it appears, i.e., $s_{\sigma_j} = j$ and $\sigma_{s_i} = i$.

a) Non interference: 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.

c) Manipulation feasibility: Each item in the packing plan must be loadable by a manipulator without disturbing previously packed items (Figs. 2.b and 2.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 $a = (0, 0, 0)$ 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 ∂C (4):

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

$$SV_i^\circ \cap \partial C = \emptyset. \tag{4}$$

B. Nondeterministic problems

Definition 3 (NDOP): The *nondeterministically ordered packing 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: $T_{\sigma_1} = \pi((), \sigma_1)$, $T_{\sigma_2} = \pi((T_{\sigma_1}), \sigma_2)$, ..., $T_{\sigma_n} = \pi((T_{\sigma_1}, \ldots, T_{\sigma_{n-1}}), \sigma_n)$.

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:

$$\pi((\sigma_1, \ldots, \sigma_{j-1}), \sigma_j) \equiv \pi((T_{\sigma_1}, \ldots, T_{\sigma_{j-1}}), \sigma_j). \tag{5}$$

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

Definition 7 (QOP): The *quasi-online packing 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).

C. Container optimization variants

Above 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:

- Offline: Find the container $C \in \mathcal{C}$ with minimum cost that yields a feasible packing plan for item set $\mathcal{I}$.
- Nondeterministically-ordered: Find the container $C \in \mathcal{C}$ with minimum cost that yields a feasible packing plan for any ordering of item set $\mathcal{I}$.
- Quasi-online: Find the container $C \in \mathcal{C}$ with 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 1 bin, a container with 2 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 non-decreasing cost until a successful packing policy is found.

IV. Method

We make use of an offline robot packing planner [19] with a small amount of modification. The responsibility of the offline planner is to generate a feasible packing plan given the constraints outlined above, while our key contributions are novel methods to invoke the planner and to validate plans under permutations of item orders.

A. Offline planner

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

$$P \leftarrow \text{Offline-Pack}(\sigma_{1:j}^{fixed}, P^{prior}, \sigma_{j+1:k}^{next}) \tag{6}$$

producing either a feasible plan $P = (\sigma_{1:n}, T_{1:n})$ or "failure." The inputs $\sigma_{1:j}^{fixed}$ specify that j items of the prior plan P^{prior} should be kept in their previous positions, and $\sigma_{j+1:k}^{next}$ 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 $\sigma_{1:j} = \sigma_{1:j}^{fixed}$, $\sigma_{j+1:k} = \sigma_{j+1:k}^{next}$, and each fixed transform T_j for $j \in \sigma_{1:k}^{fixed}$ matches the corresponding transform in P^{prior}.

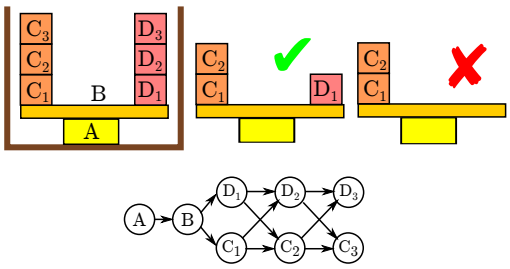


Fig. 3. A plan and its dependency graph. C_2 requires D_1 to be present to maintain the stability constraint, because otherwise the imbalanced weight on B would cause tipping. Similarly, D_2 depends on C_1, and so forth for C_3 and D_3. This CDG is compatible with orders of the form $AB(C_1D_1)(C_2D_2)(C_3D_3)$ where the (XY) denotes either XY or YX.

The offline planner used here is a constructive, heuristic method that is easily modified to handle the required changes. In Sec. V we also consider offline packing heuristics that make the job of generating a nondeterministic plan easier, but these are not strictly necessary.

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 dependency graphs* (CDGs) $G_1, \ldots, G_m$ as defined in Sec. IV-C. An individual packing plan can be used for the set of orderings that are compatible with its dependency 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 Alg. 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 dependency graphs

A fundamental data structure that will allow us to verify compatibility is the constraint dependency graph (CDG). This 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 directed acyclic graph (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 dependency 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 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 iff it does not violate any dependency in P's CDG. In other words, a feasible packing plan P with a dependency-free CDG $G = (\mathcal{I}, \emptyset)$ is compatible with all orderings. More precisely, we can state:

Lemma 1: Let $G = (\mathcal{I}, E)$ be the CDG of a feasible plan P. An ordering $\sigma'_{1:n}$ is incompatible with P iff 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 in our algorithm is to verify 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 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 dependency 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 ordering 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 $G_1, \ldots, G_m$, then v can be safely ignored. This is because v can be assigned at any point without affecting dependencies.

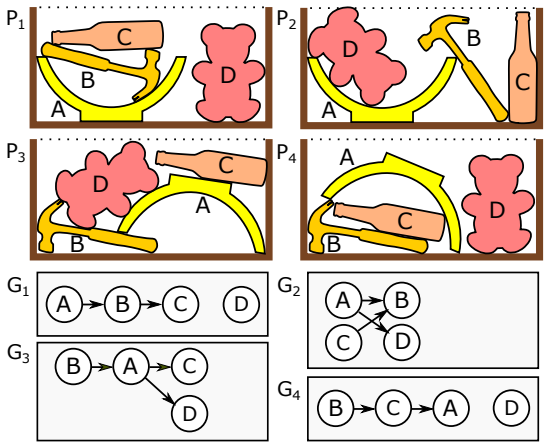


Fig. 4. A set of plans $P_1, \ldots, P_4$ (top) and their dependency graphs $G_1, \ldots, G_4$ (bottom). For any ordering beginning with A, there is at least one plan (P_1 or P_2) compatible with it. But for any ordering beginning with BDA, CB, CD, DAC, or DC, no plans are compatible.

Algorithm 1: Verify-CDG-Coverage$(\mathcal{I}, (E_1, \ldots, E_m))$

input : a set of items $\mathcal{I}$
dependency graphs $(E_1, \ldots, E_m)$

1 **if** *there exists* $v \in \mathcal{I}$ *that is not a root in any graph* $(\mathcal{I}, E_i)$ **then return** "v incompatible";
2 **if** *any* E_i *is empty,* $i = 1, \ldots, m$ **then return** "all compatible";
3 Remove all vertices v from $\mathcal{I}$ that are singletons in every graph $(\mathcal{I}, E_i)$, $i = 1, \ldots, m$;
4 **for** $v \in \mathcal{I}$ **do**
5 $\quad \mathcal{E}^v \leftarrow ()$;
6 $\quad$ **for** $i = 1, \ldots, m$ **do**
7 $\quad\quad$ **if** v *is a root in* E_i **then**
8 $\quad\quad\quad$ Append $\mathcal{E}_i$ to E^v, but with v removed;
9 $\quad$ **end**
10 $\quad r \leftarrow$Verify-CDG-Coverage$(\mathcal{I}/\{v\}, \mathcal{E}^v)$;
11 $\quad$ **if** $r =$"$\sigma_{1:j}$ *incompatible*" **then**
12 $\quad\quad$ **return** "$v, \sigma_{1:j}$ incompatible"
13 **end**
14 **return** "all compatible"

The overall algorithm is given a vertex set $\mathcal{I}$ and the edge sets $E_1, \ldots, E_m$ of the CDGs of $P_1, \ldots, P_m$ as input, and is listed in Alg. 1. The return value is either "all compatible" or a subsequence of $\mathcal{I}$ that is incompatible with every dependency 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 E^v of dependency 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.

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).

Alg. 2 solves NDOP using this subroutine.

Algorithm 2: NDOP

input : a set of items $\mathcal{I}$
output: a solution set of plans $\mathcal{P}$, or "failure"

$\mathcal{E} \leftarrow$empty-list;
$\mathcal{P} \leftarrow$empty-list;
while *true* **do**
$\quad r \leftarrow$ Verify-CDG-Coverage$(\mathcal{I}, \mathcal{E})$;
$\quad$ **if** $r =$*"all compatible"* **then return** $\mathcal{P}$;
$\quad$ Let $\sigma_{1:k}$ be the incompatible ordering in r;
$\quad P \leftarrow$Offline-Pack$(nil, nil, \sigma_{1:k})$;
$\quad$ **if** $P =$*"failure"* **then return** "failure";
$\quad$ Add P to $\mathcal{P}$;
$\quad$ Add $CDG(P)$ to $\mathcal{E}$;
end

E. Quasi-online packing

Due to the need for shared transforms, QOP is not as amenable to 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 k, which is associated with the feasible plan $P = (\sigma_{1:n}, T_{1:n})$. For all non-placed items $\sigma'_{k+1} \notin \sigma_{1:k}$, we could call:

$$P' \leftarrow \text{Offline-Pack}(\sigma_{1:k}, P, \sigma'_{k+1}). \tag{7}$$

If $P' =$"failure", then failure is returned. Otherwise, P' is associated with a new child of N in the tree corresponding to the choice σ'_{k+1}, and the search can proceed recursively. Note that if σ_{k+1} was already the $k+1$'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 dependency subgraph $CDG(\sigma_{k+1:n}, (T_{\sigma_{k+1}}, \ldots, T_{\sigma_n}))$, 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 while maintaining a list of plans $\mathcal{P}_N$ compatible with $\sigma_{1:k}$ (i.e., all T_j match, for each fixed $j \in \sigma_{1:k}$). The search proceeds to enumerate children of $\sigma_{1:k}$. For each choice σ_{k+1} and child node C, if at least one plan in $\mathcal{P}$ is compatible with σ_{k+1}, replanning is not performed, and the choice $T_{\sigma_{k+1}}$ is fixed. If multiple plans are compatible, the value of $T_{\sigma_{k+1}}$ 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 σ_{k+1} is a root of the dependency subgraph, and whose placement of σ_{k+1} matches $T_{\sigma_{k+1}}$. If no plan is compatible, 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 C. This enables subsequent siblings, siblings of parents, etc. to use P' and avoid additional planning. Pseudocode is given in Algs. 3 and 4.

Algorithm 3: QOP-Recurse(N)

input : policy tree node N

1 Let $\sigma_{1:k}$ be the sequence of packed items in N;
2 Let $\mathcal{P}_N$ be the set of compatible plans with N;
3 Let P be any plan in $\mathcal{P}_N$, or nil if $\mathcal{P}_N = \emptyset$;
4 **if** *all* $\sigma_{k+1} \notin \sigma_{1:k}$ *are roots of* P **then return** "success";
5 **for** *all items* $\sigma_{k+1} \notin \sigma_{1:k}$ **do**
6 **if** *no plan in* $\mathcal{P}_N$ *is compatible with* σ_{k+1} **then**
7 $P' \leftarrow$Offline-Pack$(\sigma_{1:k}, P, \sigma_{k+1})$;
8 **if** $P' =$*"failure"* **then return** "failure";
9 $\mathcal{P}_C \leftarrow \{P'\}$;
10 $C \leftarrow$ add-child$(N, \sigma_{k+1}, \mathcal{P}_C)$;
11 For all ancestors A of C, add P' to $\mathcal{P}_A$;
12 **else**
13 Let $T_{\sigma_{k+1}}$ be the location compatible with the most plans in $\mathcal{P}_N$;
14 $\mathcal{P}_C \leftarrow \{P' \in P \mid P'$ is compatible with $T_{\sigma_{k+1}}\}$;
15 $C \leftarrow$ add-child$(N, \sigma_{k+1}, \mathcal{P}_C)$;
16 **end**
17 **if** *QOP-Recurse*$(C, \mathcal{P}_C)$ *fails* **then return** "failure";
18 **end**
19 **return** "success"

Algorithm 4: QOP()

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

F. Analysis

Here we show that NDOP inherits the completeness properties of the offline planner, but QOP is incomplete. Even when the offline planner is incomplete, when the NDOP or QOP result is not "failure", the solution is correct. We also analyze the behavior of Verify-CDG-Coverage and demonstrate that it is NP-complete.

1) Correctness and completeness: NDOP inherits its completeness from the offline packing planner. To see this, first observe that whenever NDOP returns a solution (Line 6), this solution is correct, because for all orderings $\sigma_{1:n} \in S_n$, Alg. 1 has shown that the solution contains some plan that is compatible with $\sigma_{1:n}$. Now consider the case where NDOP returns "failure." This can only occur when Offline-Pack returns failure for a partial ordering $\sigma_{1:k}$ (Line 7). If Offline-Pack is complete, then there is indeed no solution compatible with this ordering, and hence NDOP returns failure correctly. If it is incomplete, then NDOP may return failure incorrectly.

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

Unlike NDOP, QOP 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 assumes it has control over future item ordering, as illustrated in Fig. 5. However, if it does return a solution, then this solution is feasible even if a heuristic offline planner is used.

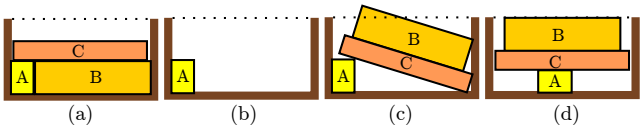


Fig. 5. An example showing that QOP (Alg 4) is not necessarily complete even with a complete offline planner. (a) The first recursive call produces a feasible plan with A placed first. (b) Once item A is placed in the planned location, the plan is infeasible for order ACB, as shown in (c). On the other hand, if A was placed as in (d), a feasible QOP solution could result.

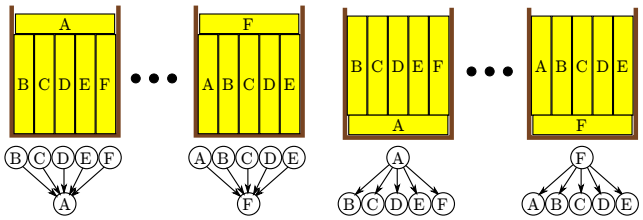


Fig. 6. (a) Worst-case behavior of Verify-CDG-Coverage occurs in an instance with n plans, where $n-1$ "books" are stacked vertically with the n'th book stacked horizontally on top. Every possible order of $k \leq n$ items is compatible with a set of $n-k+1$ plans, and a recursion depth of n is required.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

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:

Theorem 2: DEPSET-COMPAT is NP-complete.

Proof: The proof is via polynomial time reduction from 3-SAT. A 3-SAT instance consists of n Boolean variables $x_1, \ldots, x_n$ and a logical expression in disjunctive normal form, consisting of m clauses

$$(y_{11} \vee y_{12} \vee y_{13}) \wedge \cdots \wedge (y_{m1} \vee y_{m2} \vee y_{m3}) \qquad (8)$$

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 dependency 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 illustrated 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 i'th conjunctive clause in (8). 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

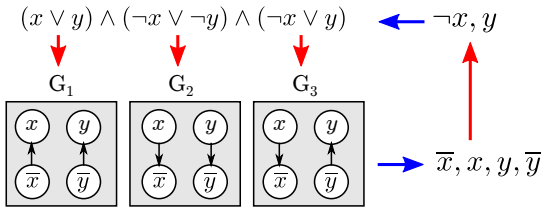


Fig. 7. Illustrating the reduction from SAT. Each clause (upper left) is converted into a dependency graph (lower left), and a counterexample (lower right) ordering corresponds to a SAT solution (upper right).

any assignment. Otherwise, we construct a dependency 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 dependency 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 $x_k \leftarrow T$, and if not, $x_k \leftarrow F$. 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 dependency 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 dependency 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 dependency graphs, e.g., separable, bipartite graphs with at most 3 dependencies! Experimentally, we have observed that DEPSET-COMPAT problems corresponding to hard 3-SAT problems (e.g., with clause-to-variable ratio of ∼4.24 [10]) also exhibit exponential complexity when solved via Alg. 1.

V. Planning Heuristics

Although DEPSET-COMPAT is NP-complete, computation time of NDOP and QOP is dominated by time spent in the offline planner, because each plan requires searching over 6D object pose. The number of plans requested, and hence 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. Dependency minimization heuristic

Constructive packing chooses an item's location based on certain placement heuristics, such as deepest-bottom-left-first (DBLF) [20], or heightmap minimization (HM) [19, 20], to maximize packing density. For nondeterministic packing, we would like to generate plans with few dependencies. We introduce a *dependency 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.

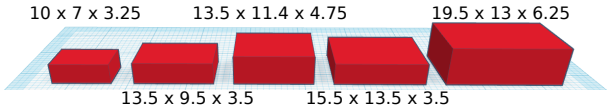


Fig. 8. The dimensions (in inches) of containers 1–5 used in our experiments, in order of increasing length + girth.

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

B. Matching prior placements

In QOP it is beneficial for the offline planner to place as many "free" items (i.e., those not in $\sigma^{fixed}_{1:j}$ or $\sigma^{next}_{j+1:k}$) 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 Alg. 2 with a fixed number of iterations, or break the recursion of QOP 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 possibly suboptimal solution.

VI. Experiments

Our experiments test the NDOP and QOP algorithms with random item sets of 3D scanned objects from the APC 2015 [17] and YCB [2] datasets (94 objects total). The containers used in these experiments are the five boxes used in the Amazon Robotics Challenge 2017 (Fig. 8), 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.12xlarge instance.

Our experiments use three testing datasets in which the item sets have different size: a) typical shopping carts of 2–5 items, b) large sets of 5 items, and c) stress tests with 10 items. For each category, we generate 1,000 random item sets by drawing items at random, and verifying with an offline planner that there exists is a feasible packing in one of

Fig. 9. An NDOP solution for packing 5 items in container 5.

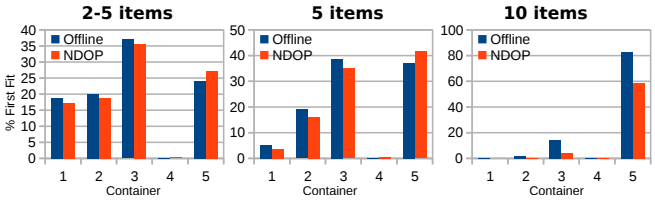
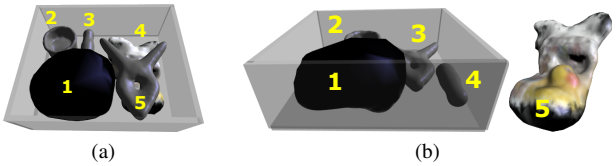


Fig. 10. Distribution of the solution container for offline / NDOP container optimization and various itemset sizes. Most small instances can be packed in any order, but with more items the variability in order requires larger boxes.

the five containers. In the 2–5 category, 250 item sets of each size are included. The results for the NDOP planner, running in container optimization mode, are summarized in Tab. I, and Fig. 9 illustrates a solution. As might be expected, the running time and the number of offline planner calls increases with the number of items, but we do not observe the exponential running time of worst-case instances. Other experiments suggest the dependency minimization heuristic reduces mean and maximum running times by approximately 20% and 50%, respectively.

Observe also that the success rate drops with increasing numbers of items, as the 10-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, so we cannot determine whether the solver is failing incorrectly. Fig. 10 shows the distribution of the minimum-cost container found. In cases with 5 or fewer items, NDOP often successfully packs in the same box as the offline planner. Observe that with 5 items, approximately 5% of test cases require container 5, even though they can be packed offline in containers 1–4. Fig. 11 illustrates an example of such a case.

Performance results for QOP in container 5 are given in Tab. II, with a representative solution shown in Fig. 12. Up



(a) (b)

Fig. 11. A failure case for packing 5 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.

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

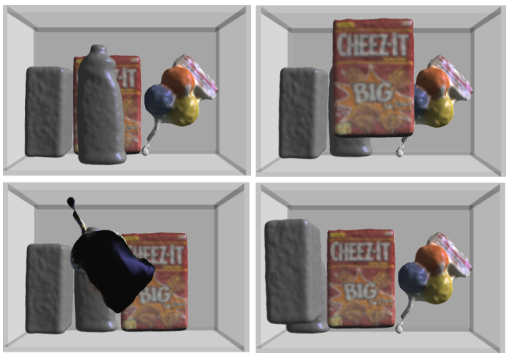


Fig. 12. A QOP solution for 4 items in container 5. Due to the matching heuristic, only four plans are needed (one for each item placed last).

to 5 items, the success rates are quite similar to NDOP, but the maximum running times and 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 is faster than NDOP on average. QOP struggles with 10 items, with a long-tailed distribution: 24 instances could not be solved within a 24-hour cutoff.

VII. Conclusion

This paper formulated two novel packing problems with nondeterministic item ordering and presented practical solvers that handle irregular 3D shapes and item sets up to size 10. 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, whether efficiency gains are possible with multiple identical 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. It may also be worth considering a *k-buffered quasi-online* variant, in which the packer has a "buffer" that can hold up to k items before packing in the ultimate container [4, 8]. A similar variant might allow items to be repacked using k hands.

Acknowledgments

The authors thank Weidong Sun and Yifan Zhu for assistance proofreading the paper. This work is supported by an Amazon Research Award.

References

[1] B. Baker, E. Coffman, Jr., and R. Rivest. Orthogonal packings in two dimensions. *SIAM Journal on Computing*, 9(4):846–855, 1980. doi: 10.1137/0209064.

[2] Berk Calli, Arjun Singh, James Bruce, Aaron Walsman, Kurt Konolige, Siddhartha Srinivasa, Pieter Abbeel, and Aaron M Dollar. Yale-cmu-berkeley dataset for robotic manipulation research. *The International Journal of Robotics Research*, 36(3):261–268, April 2017.

[3] N. Correll, K. E. Bekris, D. Berenson, O. Brock, A. Causo, K. Hauser, K. Okada, A. Rodriguez, J. M. Romano, and P. R. Wurman. Analysis and observations from the first amazon picking challenge. *IEEE Transactions on Automation Science and Engineering*, 15(1):172–188, Jan 2018. ISSN 1545-5955. doi: 10.1109/TASE.2016.2600527.

[4] János Csirik and David S Johnson. Bounded space online bin packing: Best is better than first. *Algorithmica*, 31(2):115–138, 2001.

[5] Edgar den Boef, Jan Korst, Silvano Martello, David Pisinger, and Daniele Vigo. Erratum to the three-dimensional bin packing problem: Robot-packable and orthogonal variants of packing problems. *Operations Research*, 53(4):735–736, 2005. doi: 10.1287/opre.1050.0210.

[6] Jens Egeblad. Placement of two and threedimensional irregular shapes for inertia moment and balance. *International Transactions in Operational Research*, 16:789 – 807, 06 2009.

[7] Jens Egeblad, Benny K. Nielsen, and Allan Odgaard. Fast neighborhood search for two- and three-dimensional nesting problems. *European Journal of Operational Research*, 183(3):1249 – 1266, 2007. ISSN 0377-2217. URL https://doi.org/10.1016/j.ejor.2005.11.063.

[8] Leah Epstein and Elena Kleiman. Resource augmented semi-online bounded space bin packing. *Discrete Applied Mathematics*, 157(13):2785–2798, 2009. URL https://www.sciencedirect.com/science/article/pii/S0166218X09001139.

[9] Oluf Faroe, David Pisinger, and Martin Zachariasen. Guided local search for the three-dimensional bin-packing problem. *INFORMS Journal on Computing*, 15 (3):267–283, 2003.

[10] Jon W. Freeman. Hard random 3-sat problems and the davis-putnam procedure. *Artificial Intelligence*, 81(1):183 – 198, 1996. ISSN 0004-3702. URL http://www.sciencedirect.com/science/article/pii/0004370295000518.

[11] Xin Han, Francis YL Chin, Hing-Fung Ting, Guochuan Zhang, and Yong Zhang. A new upper bound 2.5545 on 2d online bin packing. *ACM Transactions on Algorithms (TALG)*, 7(4):50, 2011.

[12] D. Johnson, A. Demers, J. Ullman, M. Garey, and R. Graham. Worst-case performance bounds for simple one-dimensional packing algorithms. *SIAM Journal on Computing*, 3(4):299–325, 1974.

[13] Thomas Kämpke. Simulated annealing: Use of a new tool in bin packing. *Annals of Operations Research*, 16 (1):327–332, Dec 1988.

[14] Xiao Liu, Jia-min Liu, An-xi Cao, and Zhuang-le Yao. Hape3d—a new constructive algorithm for the 3d irregular packing problem. *Frontiers of Information Technology & Electronic Engineering*, 16(5):380–390, May 2015. ISSN 2095-9230. doi: 10.1631/FITEE.1400421.

[15] Silvano Martello and Daniele Vigo. Exact solution of the two-dimensional finite bin packing problem. *Management Science*, 44(3):388–399, 1998. doi: 10.1287/mnsc.44.3.388.

[16] Silvano Martello, David Pisinger, and Daniele Vigo. The three-dimensional bin packing problem. *Operations Research*, 48(2):256–267, 2000. doi: 10.1287/opre.48.2.256.12386.

[17] Rutgers APC RGB-D Dataset. URL http://pracsyslab.org/rutgers_apc_rgbd_dataset. (Accessed Jan 28, 2019).

[18] Steven S Seiden. On the online bin packing problem. *Journal of the ACM (JACM)*, 49(5):640–671, 2002.

[19] Fan Wang and Kris Hauser. Stable bin packing of non-convex 3d objects with a robot manipulator. *ArXiv*, (arXiv:1812.04093 [cs.RO]), 2018. URL https://arxiv.org/abs/1812.04093.

[20] Lei Wang, Songshan Guo, Shi Chen, Wenbin Zhu, and Andrew Lim. Two natural heuristics for 3d packing with practical loading constraints. In Byoung-Tak Zhang and Mehmet A. Orgun, editors, *PRICAI 2010: Trends in Artificial Intelligence*, pages 256–267, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.