A Framework for Push-Grasping in Clutter

Mehmet R. Dogar Siddhartha S. Srinivasa

The Robotics Institute, Carnegie Mellon University
5000 Forbes Avenue, Pittsburgh, PA, USA
{mdogar, siddh}@cs.cmu.edu

***Abstract*—Humans use a remarkable set of strategies to manipulate objects in clutter. We pick up, push, slide, and sweep with our hands and arms to rearrange clutter surrounding our primary task. But our robots treat the world like the Tower of Hanoi — moving with pick-and-place actions and fearful to interact with it with anything but rigid grasps. This produces inefficient plans and is often inapplicable with heavy, large, or otherwise ungraspable objects. We introduce a framework for planning in clutter that uses a library of actions inspired by human strategies. The action library is derived analytically from the mechanics of pushing and is provably conservative. The framework reduces the problem to one of combinatorial search, and demonstrates planning times on the order of seconds. With the extra functionality, our planner succeeds where traditional grasp planners fail, and works under high uncertainty by utilizing the funneling effect of pushing. We demonstrate our results with experiments in simulation and on HERB, a robotic platform developed at the Personal Robotics Lab at Carnegie Mellon University.**

I. Introduction

Humans routinely perform remarkable manipulation tasks that our robots find impossible. Imagine waking up in the morning to make coffee. You reach into the fridge to pull out the milk jug. It is buried at the back of the fridge. You immediately start rearranging content — you push the large heavy casserole out of the way, you carefully pick up the fragile crate of eggs and move it to a different rack, but along the way you push the box of leftovers to the corner with your elbow.

The list of primitives that we use to move, slide, push, pull and play with the objects around us is nearly endless. But they share common themes. We are fearless to *rearrange clutter* surrounding our primary task — we care about picking up the milk jug, and everything else is in the way. We are acutely aware of the *consequences of our actions* — we push the casserole with enough control to be able to move it without ejecting it from the fridge.

How can we enable our robots to fearlessly rearrange the clutter around them while maintaining provable guarantees on the consequences of their actions? How can we do this in reasonable time? We would rather not have our robot stare at the fridge for 20 minutes planning intricate moves. Finally, can we demonstrate that these human-inspired actions do work on our robots with their limited sensing and actuation abilities? These are the research questions we wish to address in this paper.

The idea of rearranging objects to accomplish a task has been around for a few hundred years. We encounter this idea in games like the Tower of Hanoi [1], the 15-Puzzle and numerous others. The blocks-world problem [2] introduced this idea to the AI community. STRIPS [3] is a well-known planner to solve this problem. Planners that solve similar rearrangement problems in manipulation using real robotic hardware are also known [4]. In all of these cases, the physical act of manipulating an object is abstracted into a simple action, like pick-and-place. The problem is then reduced to a discrete search over all the actions that can be applied to all of the objects. The exact solution has been shown to have combinatorial complexity and various encouraging heuristics have been proposed. While extremely successful and algorithmically elegant, the simplified assumptions on actions severely restrict versatility. For example, such an algorithm would produce a solution whereby the robot carefully empties the contents of the fridge onto the countertop, pulls out the milk jug and then carefully refills the fridge. A perfectly valid plan, but one that is inefficient, and often impossible to execute with heavy, large, or otherwise ungraspable objects.

Pick-and-place actions are, however, easy to analyze. Once an object is rigidly grasped, it can be treated as an extension of the robot body, and the planning problem reduces to one of geometry. Performing actions other than pick-and-place requires reasoning about the consequences of actions.

A separate thread of work, rooted in Coulomb's formulation of friction, uses mechanics to analyze the consequences of manipulation actions. For example Mason [5] investigates the mechanics and planning of pushing in object manipulation under uncertainty. One of the first planners that incorporates the mechanics of pushing was developed by Lynch and Mason [6]. This planner is able to push an object in a stable manner using edge-edge contact to a goal position, using a quasi-static analysis of the mechanics of pushing. Brost [7] presents an algorithm that plans parallel-jaw grasping motions for polygonal objects with pose uncertainty. The object is pushed by one plate towards the second one, and then squeezed between the two. Howe and Cutkosky [8] show how the *limit surface*, which determines how a pushed object moves in the quasi-static mode, can be approximated by a three-dimensional ellipsoid in robotic pushing tasks.

In this work we make an attempt at merging these two threads of work: geometric planning and mechanical modeling and analysis. We present a framework that

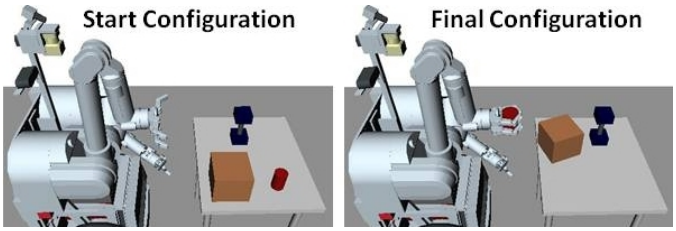


Fig. 1. An example scene. The robot's task is picking up the red can. The robot rearranges the clutter around the goal object and achieves the goal in the final configuration. The robot executes the series of actions shown in Fig. 2. We present the planning process in Fig. 3.

plans sequences of actions to rearrange clutter in manipulation tasks. This is a generalization of the planner from Stilman et al. [4]. But our framework is not restricted to pick-and-place operations and can accomodate other non-prehensile actions. We also present mechanically realistic pushing actions that are integrated into our planner.

The presented framework opens up the possibility to use different non-prehensile manipulation primitives as a part of the same planner. Researchers came up with many such primitives over the years. In previous work [9] we introduced *push-grasping* as a robust way of grasping objects under uncertainty and clutter, and it is used as one of the actions in this paper. Stulp et al. [10] present a system to learn similar action primitives to robustly grasp objects. Lynch [11] uses *toppling* as a manipulation primitive. Diankov et al. [12] use *caging* to open doors as an alternative to grasping the handle rigidly. Chang et al. [13] present a system that plans to rotate an object on the support surface, before grasping it. Omrcen et al. [14] propose a method to learn the effect of pushing actions on objects and then use these actions to bring an object to the edge of a table for successful grasping. Kappler et al. [15] propose a generic representation for such pre-grasp manipulation of objects.

Through the use of different non-prehensile actions, our planner generates plans where an ordinary pick-and-place planner cannot; e.g. when there are large, heavy ungraspable objects in the environment. We also show that our planner is robust to uncertainty.

II. Planning Framework

We present an open-loop planner that rearranges the clutter around a goal object. This requires manipulating multiple objects in the scene. The planner decides which objects to move and the order to move them, decides where to move them, chooses the manipulation actions to use on these objects, and accounts for the uncertainty in the environment all through this process. This section describes how we do that.

We describe our framework with the following example (Fig. 1). The robot's task is picking up the red can. There are two other objects on the table: a brown box which is too large to be grasped, and the dark blue dumbbell which is too heavy to be lifted.

The sequence of robot actions shown in Fig. 2 solves this problem. The robot first pushes the dumbbell away to clear a portion of the space, which it then uses to push the box into. Afterwards it uses the space in front of the red can to grasp and move it to the goal position.

Fig. 2 also shows that the actions to move objects are planned backwards in time. We visualize part of this planning process in Fig. 3. In each planning step we move a single object and plan two arm trajectories. The first one (e.g. *Push-grasp* and *Sweep* in Fig. 3) is to manipulate the object. The second one (*GoTo* in Fig. 3) is to move the arm to the initial configuration of the next action to be executed. We explain the details of these specific actions in Section III. We discuss a number of questions below to explain the planning process and then present the algorithm in Section II-E.

A. Which objects to move?

In the environment there are a set of movable objects, `obj`. The planner identifies the objects to move by first attempting to grasp the goal object (Step 1 in Fig. 3). During this grasp, both the robot and the red can, as it is moved by the robot, are allowed to penetrate the space other objects in `obj` occupy. Once the planner finds an action that grasps the red can, it identifies the objects whose spaces are penetrated by this action and adds them to a list called `move`. These objects need to be moved for the planned grasp to be feasible. At the end of Step 1 in Fig. 3, the brown box is added to `move`.

We define the operator **FindPenetrated** to identify the objects whose spaces are penetrated:

$$\textbf{FindPenetrated}(vol, \texttt{obj}) = \{\texttt{o} \in \texttt{obj} \mid vol \text{ penetrates the space of } \texttt{o}\}$$

We compute the volume of space an object occupies by taking into account the pose uncertainty (Section II-B).

In subsequent planning steps (e.g. Step 2 in Fig. 3) the planner searches for actions that move the objects in `move`. The robot and the manipulated object are again allowed to penetrate other movable objects' spaces, and penetrated objects are added to `move`.

This recursive process continues until all the objects in `move` are moved. The objects that are planned for earlier should be moved later in the execution. In other words, we do backward planning to identify the objects to move.

Allowing the planner to penetrate other objects' spaces can result in a plan where objects are moved unnecessarily. Hence, our planner tries to minimize the number of these objects. This is described in Section III.

We also restrict the plans we generate to *monotone* plans; i.e. plans where an object can be moved at most once. This avoids dead-lock situations where a plan to move object A results in object B being moved, which in turn makes object A move, and so on. But more importantly restricting the planner to monotone plans makes the search space smaller: the general problem of planning with multiple movable objects is NP-hard [16]. We enforce monotone plans by keeping a list of objects called `avoid`. At the end of each sucessful planning step the manipulated object is added to `avoid`. The planner is *not* allowed to penetrate the spaces of the objects in

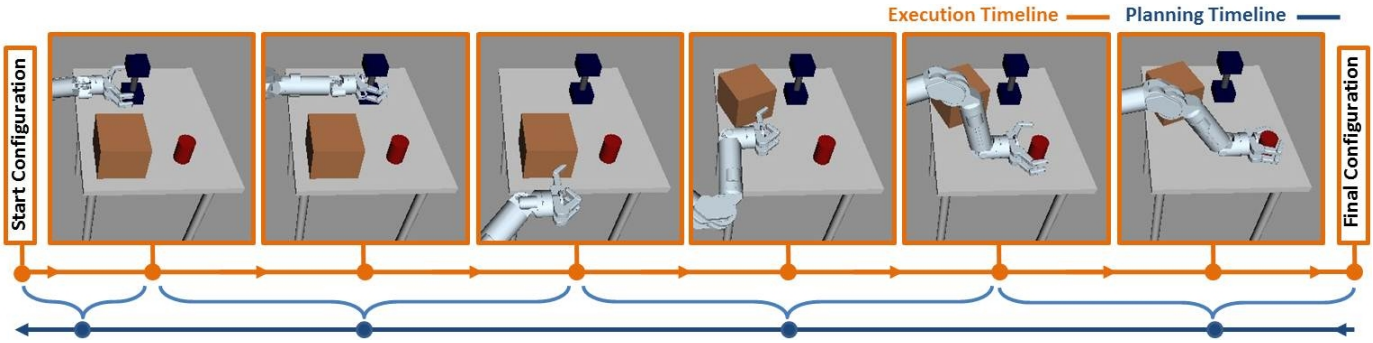


Fig. 2. We show the snapshots of the planned actions in the order they are executed. The execution timeline goes from left to right. Each dot on the execution timeline corresponds to a snapshot. Planning goes from right to left. Each dot on the planning timeline corresponds to a planning step. The connections to the execution timeline shows the robot motions planned in a planning step. Details of this planning process are in Fig. 3.

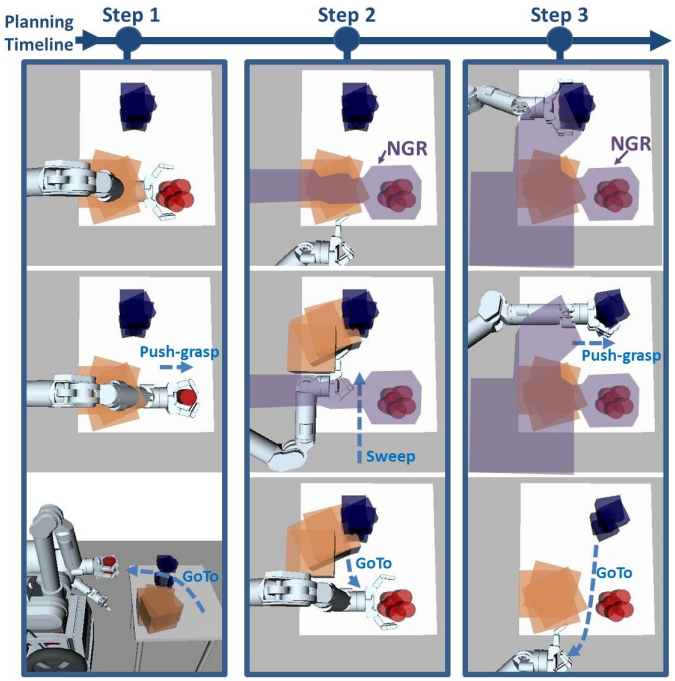


Fig. 3. The planning timeline. Three snapshots are shown for each planning step. The planner plans two consecutive arm motions at each step, from the first snapshot to the second snapshot, and from the second snapshot to the third snapshot. These motions are represented by blue dashed lines. The purple regions show the *negative goal regions* (*NGR*s), which are the regions the object needs to be moved out of (Section II-D). The object pose uncertainty is represented using a collection of samples of the objects.

`avoid`. In Fig. 3 in Step 2 the `avoid` list includes the red can, in Step 3 it includes the red can and the brown box.

B. How to address uncertainty?

Robots can detect and estimate the poses of objects with a perception system [17]. Inaccuracies occur in pose estimation, and manipulation plans that do not take this into account can fail. Non-prehensile actions can also decrease or increase object pose uncertainty. Our planner generates plans that are robust to uncertainty. We explicitly represent and track the object pose uncertainty during planning. Fig. 3 visualizes the pose uncertainty using copies of the object at different poses.

In this paper we use the word *region* to refer to a subset of the configuration space of a body. We define the *uncertainty region* of an object o at time t as the set of poses it can be in with probability larger than ϵ:

$$U(o,t) = \{q \in \mathbb{SE}(3) | o \text{ is at } q \text{ at time } t \text{ with prob.} > \epsilon\}$$

The manipulation actions change the uncertainty of an object. This is represented as a trajectory ν:

$$\nu : [0,1] \rightarrow \mathcal{R}$$

where $\mathcal{R}$ is the set of all subsets of $\mathbb{SE}(3)$. We call ν the *evolution of the uncertainty region* of that object.

In the rest of this paper, we will drop the time argument to U and use $U(o)$ to stand for the initial uncertainty region (i.e. the uncertainty region before manipulation) of the object o. We will use ν to refer to the uncertainty region as the object is being manipulated, and specifically $\nu[1]$ to refer to the final uncertainty region of the object after manipulation. We get $U(o)$ by modeling the error profile of our perception system. Each manipulation action outputs ν. Section III describes how this is computed for our actions.

During planning, we compute the volume of space an object occupies using U, not only the most likely pose. Likewise we compute the space swept by a manipulated object using ν. We define the operator **Volume**, which takes as input an object and a region, and computes the total 3-dimensional volume of space the object occupies if it is placed at every point in the region. For example, **Volume**$(o, U(o))$ gives the volume of space occupied by the initial uncertainty region of object o. We overload **Volume** to accept trajectories of regions too; e.g. **Volume**(o, ν) gives the volume of space swept by the uncertainty of the object during its manipulation.

C. How to move an object?

The traditional manipulation planning algorithms assume two types of actions: *Transfer* and *Transit* [18], [19] or *Manipulation* and *Navigation* [4]. Transit does not manipulate any objects, Transfer manipulates only an already rigidly grasped object. Our algorithm lifts this assumption and opens the way for non-prehensile actions. At each planning step, our planner searches over a set of possible actions in its action library. For example

in Step 1 of Fig. 3 the planner uses the action named *push-grasp*, and in Step 2 it uses the action *sweep*. *Push-grasp* uses pushing to funnel a large object pose uncertainty into the hand. *Sweep* uses the outside of the hand to push large objects. We will describe the details of specific actions we use (e.g. push-grasp and sweep) in Section III. Below we present the general properties an action should have so that it can be used by our planner.

In grasp based planners robot manipulation actions are simply represented by a trajectory of the robot arm: $\tau : [0,1] \rightarrow \mathcal{C}$ where $\mathcal{C}$ is the configuration space of the robot. The resulting object motion can be directly derived from the robot trajectory. With non-prehensile actions this is not enough and we also need information about the trajectory of the object motion: the evolution of the uncertainty region of the object. Hence the interface of an action `a` in our framework takes as an input the object to be moved `o`, a region of goal configurations for the object G, and a volume of space to avoid *avoidVol*; and outputs a robot trajectory τ, and the evolution of the uncertainty region of the object during the action ν:

$$(\tau, \nu) \leftarrow a(\mathtt{o}, G, avoidVol) \tag{1}$$

The returned values τ and ν must satisfy:

- $\nu[1] \subseteq G$; i.e. at the end all the uncertainty of the object must be inside the goal region.
- $\textbf{Volume}(\mathtt{robot}, \tau)$ and $\textbf{Volume}(\mathtt{o}, \nu)$ are collision-free w.r.t *avoidVol*; where `robot` is the robot body.

If the action cannot produce such a τ and ν, it returns an empty trajectory, indicating failure.

We also use a special action called *GoTo*, that does not necessarily manipulate an object, but moves the robot arm from the end of one object manipulation action to the start of other.

D. Where to move an object?

The planner needs to decide where to move an object — the goal of the action. This is easy for the original goal object, the red can in the example above. It is the goal configuration passed into the planner, e.g. the final configuration in Fig. 1. But for subsequent objects, the planner does not have a direct goal. Instead the object (e.g. the box in Step 2 of Fig. 3) needs to be moved out of a certain volume of space in order to make the previously planned actions (Step 1 in Fig. 3) feasible. We call this volume of space the *negative goal region* (*NGR*) at that step (shown as a purple region in Fig. 3) [1]. Given an *NGR* we determine the goal G for an object `o` by subtracting the *NGR* from all possible stable poses of the object in the environment: $G \leftarrow \textbf{StablePoses}(\mathtt{o}) - NGR$.

The *NGR* at a planning step is the sum of the volume of space used by all the previously planned actions. This includes both the space the robot arm sweeps and the space the manipulated objects' uncertainty regions sweep. At a given planning step, we compute the negative goal region to be passed on to the subsequent planning step, NGR_{next}, from the current *NGR* by:

$$NGR_{next} \leftarrow NGR + \textbf{Volume}(\mathtt{robot}, \tau) + \textbf{Volume}(\mathtt{o}, \nu)$$

where τ is the planned robot trajectory, `o` is the manipulated object, and ν is the evolution of the uncertainty region of the object at that planning step.

[1]Note that the *NGR* has a 3D volume in space. In Fig. 3 it is shown as a 2D region for clarity of visualization.

E. Algorithm

In our problem, a robot whose configurations we denote by $r \in \mathcal{C} \subseteq \mathbb{R}^n$ interacts with movable objects in the set `obj`. We wish to generate a sequence of robot motions `plan` that brings a goal object $\mathtt{goal} \in \mathtt{obj}$ into a goal pose $q_{\mathtt{goal}} \in \mathbb{SE}(3)$. The planning process is initiated with the call:

$$\mathtt{plan} \leftarrow \text{Reconfigure}(\mathtt{goal}, \{q_{\mathtt{goal}}\}, \{\}, \{\}, *)$$

The $*$ here means that the final configuration of the arm does not matter as long as the object is moved to $q_{\mathtt{goal}}$.

Each recursive call to the Reconfigure function is a planning step (Alg. 1). The function searches over the actions in its action library between lines 1-21, to find an action that moves the goal object to the goal configuration (line 4), and then to move the arm to the initial configuration of the next action (line 7). On line 11 it computes the total volume of space the robot and the manipulated object uses during the action. Then it uses this volume of space to find the objects whose spaces have been penetrated and adds these objects to the list `move` (line 12). If `move` is empty the function returns the plan. On line 15 the function adds the volume of space used by the planned action to the *NGR*. On line 16 it adds the current object to `avoid`. Between lines 17-20 the function iterates over objects in `move` making recursive calls. If any of these calls return a plan, the current trajectory is added at the end and returned again (line 20). The loop between 17-20 effectively does a search over different orderings of the objects in `move`. If none works, the function returns an empty plan on line 22, indicating failure, which causes the search tree to bactrack. If the planner is successful, at the end of the complete recursive process `plan` includes the trajectories in the order that they should be executed.

III. Action Library

In this section we describe the actions implemented in our action library. There are four actions.

- *Push-grasp*: Grasp objects even when they have large initial uncertainty regions.
- *Sweep*: Push objects with the outer side of the hand. Useful to move large objects.
- *GoTo*: Moves from a robot configuration to another.
- *PickUp*: Combination of Push-Grasp and GoTo. Used to grasp an object and move it to somewhere else by picking it up.

The generic interface for actions is given in (1). In this section we describe how the actions we implemented satisfy this interface.

Each action can be parametrized in different ways in a given environment. For example the robot can Push-Grasp an object by pushing in different directions. An

Algorithm 1:
$\texttt{plan} \leftarrow$ **Reconfigure**$(\texttt{o}, G, NGR, \texttt{move}, \texttt{avoid}, r^{t+2})$

1 **repeat**
2 $\texttt{a} \leftarrow$ next action from action library
3 $avoidVol \leftarrow \sum_{\texttt{o}_\texttt{i} \in \texttt{avoid}}(\textbf{Volume}(\texttt{o}_\texttt{i}, U(\texttt{o}_\texttt{i})))$
4 $(\tau_1, \nu) \leftarrow \texttt{a}(\texttt{o}, G, avoidVol)$
5 **if** τ_1 *is* empty **then**
6 Continue at line 2
7 $\tau_2 \leftarrow$ GoTo$(\tau_1[1], r^{t+2}, avoidVol + \textbf{Volume}(\texttt{o}, \nu[1]))$
8 **if** τ_2 *is* empty **then**
9 Continue at line 2
10 $\tau \leftarrow \tau_1 + \tau_2$
11 $vol \leftarrow \textbf{Volume}(\texttt{robot}, \tau) + \textbf{Volume}(\texttt{o}, \nu)$
12 $\texttt{move}_\texttt{next} \leftarrow \texttt{move} + \textbf{FindPenetrated}(vol, \texttt{obj})$
13 **if** $\texttt{move}_\texttt{next}$ *is* empty **then**
14 **return** $\{\tau\}$
15 $NGR_{next} \leftarrow NGR + vol$
16 $\texttt{avoid}_\texttt{next} \leftarrow \texttt{avoid} + \{\texttt{o}\}$
17 **foreach** $\texttt{o}_\texttt{i} \in \texttt{move}_\texttt{next}$ **do**
18 $\texttt{plan} \leftarrow \textbf{Reconfigure}(\texttt{o}_\texttt{i}, \textbf{StablePoses}(\texttt{o}_\texttt{i}) - NGR_{next}, NGR_{next}, \texttt{move}_\texttt{next} - \{\texttt{o}_\texttt{i}\}, \texttt{avoid}_\texttt{next}, \tau[0])$
19 **if** plan *is not* empty **then**
20 **return** $\texttt{plan} + \{\tau\}$
21 **until** *all actions in action library are tried*
22 **return** empty

action searches over its parameter space to find valid robot and object trajectories. In this section we specify these parameters for each action and present the way the search is done. We also explain how we compute the evolution of the uncertainty region.

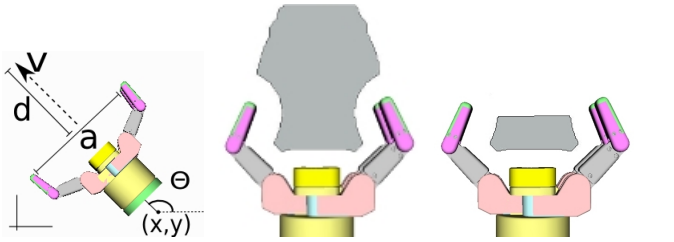
A. Push-Grasp

Our planner uses the *push-grasp* action to grasp objects. This action, which utilizes the mechanics of pushing, was introduced in [9].

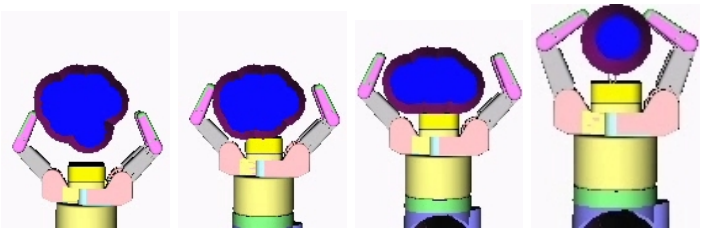
Push-grasping is a robust way of grasping objects under uncertainty. It is a straight motion of the hand parallel to the pushing surface along a certain direction, followed by closing the fingers. In effect, a push-grasp sweeps a region on the pushing surface, so that wherever an object is in that region, at the end of the push it ends up inside the hand, ready to be grasped. An example is presented in Fig. 4(d-g).

A push-grasp is parametrized (Fig. 4a) by $PG(p_h, a, d)$:

- $p_h = (x, y, \theta) \in SE(2)$ is the *initial pose* of the hand relative to the pushing surface.
- a is the *aperture* of the hand during the push. The hand is shaped symmetrically and is kept fixed during motion.
- v is the *pushing direction* along which the hand moves in a straight line. The pushing direction is normal to the palm and is fully specified by p_h.



(a) Parametrization (b) Capture Region (c) Object poses after the push



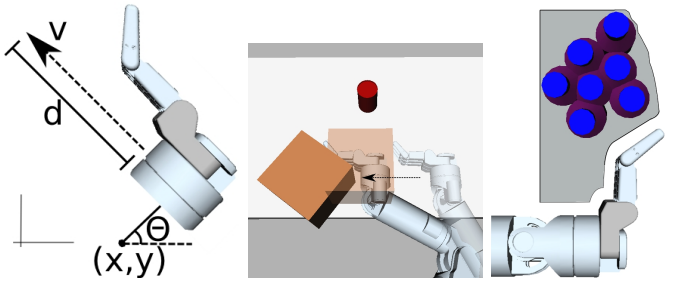
(d) Uncertainty (e) Push grasp (f) Push grasp (g) Push grasp

Fig. 4. (a) Push-grasp parametrization. (b) The capture region of a for a rotationally symmetric bottle. Every point corresponds to a bottle position where the coordinate frame of the bottle is at its center. (c) Uncertainty region of the object after the push, before closing the fingers. (d-g) A push grasp funneling the uncertainty into the hand.

- d is the *push distance* measured as the translation of the hand along the pushing direction.

We do not search all the parametrization space since most of the space produces push-grasps that do not even touch the object. Instead, we discretize the directions to push the object at a resolution of $\pi/18$ rad (i.e. 36 different directions) and use a predefined set of 9 hand aperture values (some of which may not be used if the object does not fit into the hand with that aperture). For each value of v and a in a simulated environment we place the hand over the most probable object position with the direction v and aperture a. Then we move the hand along the line perpendicular to v for an amount l, the *lateral offset*. l changes between $[-a, a]$ with a resolution of $0.01m$. After this we move the hand backwards (in the direction of $-v$) until the hand is not penetrating the initial uncertainty region of the object. At the end the hand is at a specific p_h. We then use the *capture regions* (described below) to decide if all the uncertainty of the object can be funnelled into the hand, and if yes, to compute the necessary pushing distance d to do that.

For a given object, the *capture-region* (Fig. 4b) of a parametrized push-grasp is the set of all object poses that results in a successful grasp. We denote a capture region by $C(PG, \texttt{o})$, where PG is a parametrized push-grasp, and $\texttt{o}$ is the object. We use our pushing simulation to compute the set of poses the object can be at, so that, at the end of the push the object ends up inside the hand (Fig. 4c), ready to be grasped. We do not assume that we know all the necessary physical properties of the object. Instead, we assume conservative values for these parameters, such that the computed capture region will still be valid for any other reasonable choice of these values. We do not do this simulation during planning.



(a) Parametrization (b) Example sweep (c) Capture Region

Fig. 5. (a) Sweep is parametrized by the initial hand pose and pushing distance. (b) Sweeping can move objects that are too large to be grasped. (c) The capture region of the sweep action for the a cylindrically symmetric bottle.

We precompute the the necessary information to build a capture region offline, and during planning we can build capture regions for different values of a and d in a fast way. A push-grasp funnels the initial uncertainty region of an object into the hand if the uncertainty region is encapsulated in the capture region of the push-grasp. We use this method to find the smallest capture region (which corresponds to the shortest pushing distance d) that will succeed. More details are given in [9]. Once we know d we compute the pose the object will end up in if the push-grasp is executed and fingers are closed. We check if this pose is inside the goal region. If it is, we try generating a series of arm configurations, τ, for the push-grasp. We also make sure that the $\textbf{Volume}(\texttt{robot}, \tau)$ and $\textbf{Volume}(\texttt{o}, \nu)$ (how we compute ν is explained below) are not penetrating *avoidVol*.

While the action we compute avoids the objects in `avoid` it is allowed to penetrate the space of other movable objects as explained in Section II-A. But we try to minimize the number of such objects to get more efficient plans. Therefore we compute a heuristic value for the 36 different directions to push-grasp the object. We rotate the robot hand around the goal object and check the number of objects it collides with. We prefer directions v with a smaller number of colliding objects.

We also use the capture region (e.g. Fig. 4b) to represent the evolution of the uncertainty region, ν. As the push proceeds, the top part of the capture region shrinks towards the hand and the resulting uncertainty region is captured inside the hand (Fig. 4c). Since the object cannot escape out of the capture region during the push-grasp, the uncertainty during the action can be conservatively estimated using the shrinked capture region at every discrete step. These series of capture regions can be used to represent ν. **Volume** operator samples poses from a capture region to compute the total volume.

B. Sweep

Sweep is another action we use to move obstacles out of negative goal regions. Sweep uses the outside region of the hand to push an object. Sweeping can move objects that are too large to be grasped (Fig. 5b). Similar to Push-Grasp, we parametrize a Sweep by $S(p_h, d)$; the hand pose and the pushing distance (Fig. 5a).

A push-grasp requires a minimum pushing distance because it has to keep pushing the object until it completely rolls into the hand. Since sweeping only needs to move an object out of a certain volume of space, it does not require a particular pushing distance. But we still use the capture region to guarantee that the object will not escape the push by rolling outside during the sweep. When computing the capture region for sweep (Fig. 5c) we use the pushing simulation for the side of the fingers but approximate the other side with a straight line located at the end of the wrist link.

The sweep action can also address initial object pose uncertainty. Similar to Push-Grasp, we check that the capture region of the Sweep includes all the poses sampled from the uncertainty region of the object (Fig. 5c).

We cannot know the exact location of the object after the sweep because a sweep action does not have a particular minimum pushing distance. We know that the object ends up inside the hand at the end of a push-grasp, and the uncertainty is very small. However, for sweeping this uncertainty can be large. We approximate the evolution of the uncertainty region of sweep by using samples from two different regions. The first region is object's initial uncertainty region. Until the sweeping hand makes a contact with a sample from this region that sample is included in ν. The second region is around the sweeping surface of the hand representing all possible poses of the object in contact with the hand surface.

C. GoTo

The GoTo action moves the robot arm from one configuration to the other. The search space of the GoTo action is the configuration space of the arm. We use the Constrained Bi-directional RRT planner (CBiRRT) [20] to implement this action.

The GoTo action either does not manipulate an object or moves an already grasped object. At the end the object pose is derived from the forward kinematics of the arm.

D. PickUp

In highly cluttered environments, moving objects locally may not be possible because all the immediate space is occupied. In such cases, picking up an obstacle object and moving it to some other surface may be desirable. We implement this action in our planner as the PickUp action. PickUp is also useful to move the original goal object of the plan to the final goal configuration. We implement PickUp as a Push-Grasp followed by a GoTo.

IV. Implementation and Results

A. Implementation

We implemented the planner on our robot HERB [21]. We conducted simulation experiments using OpenRAVE [22]. We created scenes in simulation and in real world. The robot's goal was to retrieve objects from the back of a cluttered shelf and from a table. We used everyday objects like juice bottles, poptart boxes, coke cans. We also used large boxes which the robot cannot grasp.

TABLE I
PLANNING TIME COMPARISON

	Total	GT	PU	SW	PG
Pushing	25.86	10.92	6.76	6.08	1.92
Pick-and-Place	12.52	6.54	5.98	-	-

We present snapshots from our experiments in the figures of this section. The video versions can be viewed at www.cs.cmu.edu/~mdogar/pushclutter

B. Pushing vs. Pick-and-Place

Here, we compare our planner in terms of the efficiency (planning and execution time) and effectiveness (whether the planner is able to find a plan or not) with a planner that can only perform pick-and-place operations. To do this, we used our framework algorithm to create a second version of our planner, where the action library consisted of only the PickUp and GoTo actions, similar to the way traditional planners are built using *Transfer* and *Transit* operations. We modified the PickUp action for this planner, so that it does not perform the pushing at the beginning, instead it grasps the object directly. We call this planner the *pick-and-place planner*, and our original planner the *pushing planner*.

An example scene where we compare these two planners is given in Fig. 6. The robot's goal is to retrieve the coke can from among the clutter. We present the plans that the two different planners generate. The pushing planner sweeps the large box blocking the way. The pick-and-place planner though cannot grasp and pick up the large box, hence needs to pick up two other objects and avoid the large box. This results in a longer plan, and a longer execution time for the pick-and-place planner. The planning time for the pick-and-place planner is also longer, since it has to plan more actions. These times are shown on the figure.

In the previous example the pick-and-place planner was still able to generate a plan. Fig. 7 presents a scene where the pick-and-place planner fails. The pushing planner generates a plan and is presented in the figure.

C. Addressing uncertainty

One of the advantages of using pushing is that pushing actions can account for much higher uncertainty than direct grasping approaches. To demonstrate this we created scenes where we applied high uncertainty to the detected object poses. Fig. 8 presents an example scene. Here the objects have an uncertainty region which is a Gaussian with $\sigma_{x,y} = 2cm$ for translation and $\sigma_\theta = 0.05rad$ for the rotation of the object. The pick-and-place planner fails to find a plan in this scene too, as it cannot find a way to guarantee the grasp of the objects with such high uncertainty. The pushing planner generates plans even with the high uncertainty.

D. Effect on planning time

We also conducted experiments to see the effect of adding the pushing actions to the pick-and-place planner in cases where both planners would work. We created five random scenes with different graspable objects and generated plans using the pushing and pick-and-place planner in these scenes. We ran each planner three times for each scene, due to the random components of our GoTo and PickUp actions. The average planning time for each planner is shown in Table I in seconds. The division of this time to each action is also shown (GT: GoTo, PU: pick-up, SW: sweep, PG: push-grasp).

It is seen that, on average, the pushing planner takes two times the time the pick-and-place planner takes. This is due to two reasons. First reason is the time the Push-Grasp and Sweep actions take. We see that on average these two actions take 30% of the time in each planning cycle for the pushing planner. The second reason is the large uncertainty region the Sweep action generates. This usually causes more objects to be moved, which is reflected in the higher time spent on the navigate action.

V. CONCLUSION AND DISCUSSION

In this paper we present a planning framework capable of incorporating actions beyond the traditional pick-and-place operations. We introduce pushing actions that can be used to manipulate otherwise ungraspable objects. We demonstrate that this planner generates plans where a pick-and-place planner fails.

However, there are also limitations of using pushing actions that result in high uncertainty. One problem that we came across in tight spaces, e.g. shelves, was the consumption of the space by the resulting uncertainty of a sweep action. This large uncertainty sometimes also cause the unnecessary displacement of objects.

We believe some of these problems can be solved by interleaving planning with sensing. Currently we generate a sequence of actions that are executed without any sensing. Sensing can be used (i) at the level of pushing actions using tactile/force sensors; and (ii) using robot vision to look at the scene between steps of a plan.

ACKNOWLEDGMENTS

We are grateful to the members of the Personal Robotics Lab at Carnegie Mellon University. This material is based upon work supported by DARPA-BAA-10-28, NSF-IIS-0916557, and NSF-EEC-0540865. Mehmet Dogar is partially supported by a Fulbright Fellowship.

REFERENCES

[1] G. Chartrand, *Introductory Graph Theory*, New York: Dover, 1985, ch. 6.3, pp. 135–139.
[2] T. Winograd, "Procedures as a Representation for Data in a Computer Program for Understanding Natural Language," MIT, Tech. Rep. MAC-TR-84, 1971.
[3] R. E. Fikes and N. J. Nilsson, "Strips: A new approach to the application of theorem proving to problem solving," *Artificial Intelligence*, vol. 2, no. 3-4, pp. 189–208, 1971.
[4] M. Stilman, J.-U. Schamburek, J. Kuffner, and T. Asfour, "Manipulation planning among movable obstacles," in *IEEE ICRA*, 2007.
[5] M. T. Mason, "Mechanics and Planning of Manipulator Pushing Operations," *IJRR*, vol. 5, no. 3, pp. 53–71, 1986.
[6] K. M. Lynch and M. T. Mason, "Stable Pushing: Mechanics, Controllability, and Planning," *IJRR*, vol. 15, no. 6, pp. 533–556, 1996.
[7] R. C. Brost, "Automatic grasp planning in the presence of uncertainty," *IJRR*, vol. 7, no. 1, 1988.

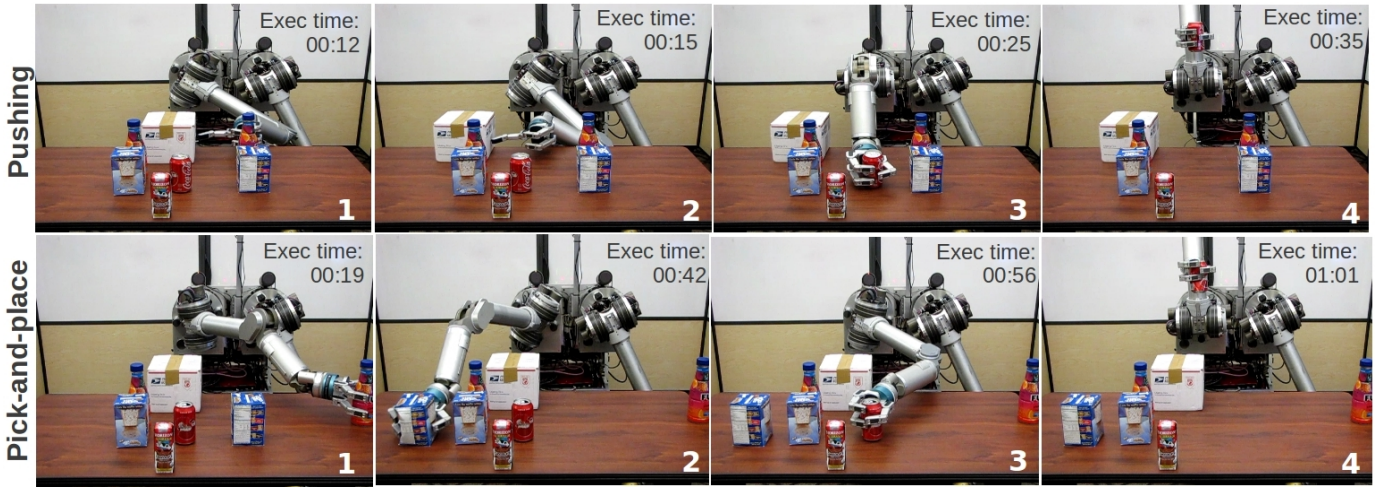


Fig. 6. The plans that the *pushing planner* and the *pick-and-place* planner generates in the same scene are presented. The pushing planner is more efficient as it is able to sweep the large box to the side. The pick-and-place plan needs to move more objects and takes more time to execute. The planning time is also more for the pick-and-place planner (27.8 sec vs. 16.6 sec) as it needs to plan more actions.

Fig. 7. An example plan to retrieve a can hidden behind a large box on a shelf. The pick-and-place planner fails to find a plan in this scene. But pushing planner finds the presented plan. Planning time is 51.2 sec.

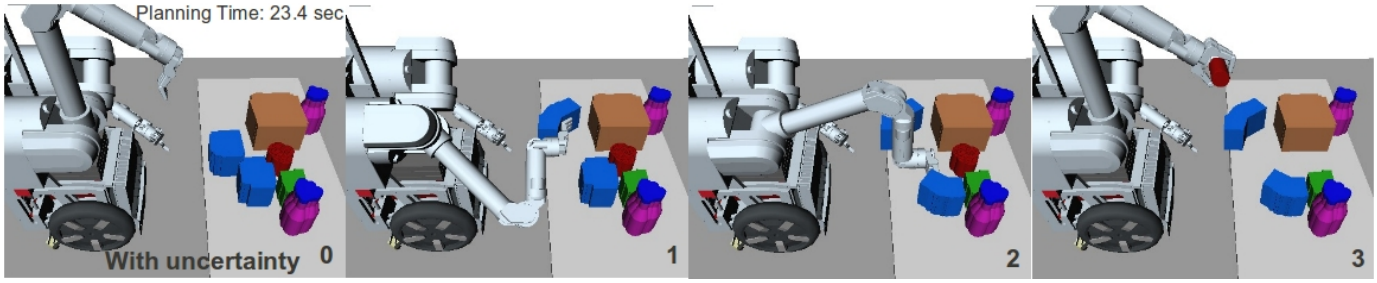


Fig. 8. An example plan under high uncertainty and clutter. The pick-and-place planner fails to find a plan in this scene, as it cannot find a feasible grasp of the objects with such high uncertainty. Pushing planner suceeds in generating a plan, presented above.

[8] R. D. Howe and M. R. Cutkosky, "Practical Force-Motion Models for Sliding Manipulation," *IJRR*, vol. 15, no. 6, pp. 557–572, 1996.

[9] M. Dogar and S. Srinivasa, "Push-Grasping with Dexterous Hands," in *IROS*, October 2010.

[10] F. Stulp, E. Theodorou, J. Buchli, and S. Schaal, "Learning to grasp under uncertainty," in *IEEE ICRA*, 2011.

[11] K. M. Lynch, "Toppling Manipulation," in *IEEE/RSJ IROS*, 1999, pp. 152–159.

[12] R. Diankov, S. Srinivasa, D. Ferguson, and J. Kuffner, "Manipulation Planning with Caging Grasps," in *Humanoids*, December 2008.

[13] L. Y. Chang, S. Srinivasa, and N. Pollard, "Planning pre-grasp manipulation for transport tasks," in *IEEE ICRA*, May 2010.

[14] D. Omrcen, C. Boge, T. Asfour, A. Ude, and R. Dillmann, "Autonomous acquisition of pushing actions to support object grasping with a humanoid robot," in *IEEE-RAS Humanoids*, December 2009, pp. 277 –283.

[15] D. Kappler, L. Chang, M. Przybylski, N. Pollard, T. Asfour, and R. Dillmann, "Representation of Pre-Grasp Strategies for Object Manipulation," in *IEEE-RAS Humanoids*, December 2010.

[16] G. Wilfong, "Motion planning in the presence of movable obstacles," in *Proceedings of the Fourth Annual Symposium on Computational Geometry*, 1988, pp. 279–288.

[17] M. Martinez, A. Collet, and S. Srinivasa, "MOPED: A Scalable and Low Latency Object Recognition and Pose Estimation System," in *IEEE ICRA*, 2010.

[18] T. Siméon, J.-P. Laumond, J. Cortés, and A. Sahbani, "Manipulation Planning with Probabilistic Roadmaps," *IJRR*, vol. 23, no. 7-8, 2004.

[19] M. Stilman and J. J. Kuffner, "Planning among movable obstacles with artificial constraints," in *In WAFR*, 2006, pp. 1–20.

[20] D. Berenson, S. Srinivasa, D. Ferguson, and J. Kuffner, "Manipulation Planning on Constraint Manifolds," in *IEEE ICRA*, May 2009.

[21] S. S. Srinivasa, D. Ferguson, C. J. Helfrich, D. Berenson, A. Collet, R. Diankov, G. Gallagher, G. Hollinger, J. Kuffner, and M. V. Weghe, "HERB: a home exploring robotic butler," *Autonomous Robots*, 2009.

[22] R. Diankov and J. Kuffner, "OpenRAVE: A Planning Architecture for Autonomous Robotics," Robotics Institute, Tech. Rep. CMU-RI-TR-08-34, July 2008.