# Planning Push and Grasp Actions: Experiments on the iCub Robot

Rui Coelho, Alexandre Bernardino
IST, Technical University of Lisbon
rui.coelho@ist.utl.pt, alex@isr.ist.utl.pt

***Abstract*—The table-top manipulation of objects is a common scenario for a robot armed with a gripper. Sometimes, grasping an object is not possible, but pushing is. As such, a robot should be able to perform both kinds of actions while manipulating different objects. The purpose of this work is to blend pushing and grasping into a planning algorithm capable of effectively computing a set of actions that accomplishes the desired task, while coping with real-time constraints, uncertainty on the outcome of push actions, using previously acquired knowledge on the dynamics of objects and learning from experience to improve its performance.**

**The representation of the state-space presented uses concepts of pushability and graspability to assert if a push or grasp can be performed at a certain poses. It also encompasses information on the effect of non-deterministic actions performed on different parts of the object to better plan the actions to take.**

**To be able to handle discretization of large state-spaces, a *Rapidly-Exploring Random Tree* (RRT) approach is used in the planner to build a path between initial and target object poses using push action exclusively. Using the pushability and graspability of the robot on a given environment, the planner can compute a path merging the two types of actions.**

**The pushing part of the planning algorithm was tested in the iCub robot and was proven to generate a possible path of poses of the gripper to be executed by the robot to take an object from one pose to another.**

***Index Terms*—grasp, object manipulation, push, reachability, RRT**

## I. Introduction

Object manipulation by a robotic arm poses an interesting subject, due to the resemble it has with the way humans interact with every day objects. Grasping and pushing objects are the most common actions in this context, which have been treated as separate problems, although they should be used together to perform a certain task.

Pushing presents itself as a more intricate problem than grasping, as for grasping, after the object has been grabbed, it becomes a matter of solving a geometric problem. However, pushing an object carries uncertainty to the resulting state of the object. It is, nonetheless, important to be able to incorporate these two actions in the planning. For instance, a push action can be used to adjust to pose of an object so that it can be easily grasped, or maybe it is faster to just push an object than performing pick and place.

To model such actions, in [1] a sampling-based motion planning approach is used, by creating a tree of possible outcomes for the push action, using a method based on rapidly-exploring random trees (RRT), for a mobile robot. The RRT approach represents an elegant solution to the problem of discretizing large state-spaces and modeling uncertainty. It successfully creates a trajectory that takes objects to the desired position by pushing only, but it lacks generalization of behaviour for similar objects. In [2], the RRT is also used in a similar fashion, although the object being used must be perfectly known through its CAD model and a physics simulator is used to determine the outcome of a push.

The approach here presented is also based on the RRT method to plan push actions, but it also addresses the problem of categorizing similar objects to generalize the result of actions performed on them. The planning algorithm also incorporates grasping, which, together with push actions, can perform a vaster array of tasks. It uses the concept of reachability [3] for each kind of action to quickly exclude impossible poses of the gripper of the planning stage, creating a reliable plan suitable for real-time operation.

The purpose of this work is, then, to present a planning algorithm capable of blending push and grasp actions in the execution, while using information on the limitations of robot, modeling the uncertainty of the effect of the actions on the object to be used to achieve a possible plan. By categorizing the objects, the effect actions have on them can be generalized to other objects, providing a tool for learning the behaviour of objects when interacted.

The paper outline is as follows: section II describes the representation of the state-space used, namely, the environment, the object and the gripper, section III describes the how the learning process is executed, section IV describes the planning algorithm, then in section V the results from the pushing algorithm implemented in the iCub robot [4] are presented and finally in section VI, final remarks and ideas for future work are given.

## II. State-Space Representation

In a typical table-top object manipulation problem, there are a few components common to all of them. There is an environment, which could be an empty table or with obstacles, where the object is manipulated. Plus, one or more objects may be placed in this environment, which should be manipulated in some way by a gripper belonging to a robot.

The state of an object consists on a position and orientation, but also on other properties important to model, such as its dimensions and the result of acting on different points of the object.

The gripper can also take many different orientations and poses on the table, though it is constrained by its own physical limitation, which is important to take into account when planning the manipulation of objects.

In this section, an efficient way to represent these three components and how these representations are interconnected is exposed.

### A. Environment

The world environment is composed of a table, objects placed on it and a gripper. The table surface is discretized in a fixed-sized grid. Each 3D cell is $3cm$ high. The width and length of the cells is established according to the dimensions of the table, so that all the table is covered by this discretization. So, the number of cells in each dimension is given by its length divided by 3 ($cm$), which is then used to computed the cell's width and length. The number of cells in the Z dimension is fixed to 10.

Each cell is assigned with a pushability and a graspability vector. These two vectors translate the ability of the gripper to reach that cell in each one of the defined discrete orientations and then perform either a push or a grasp.

To initialize the pushability and graspability vectors, the Inverse Kinematics at each cell is computed, with each one of the possible gripper's orientation (72 values are defined). Then, for the pushability affordance, only if there is a path between the current pose and one on a fixed distance away along the direction of the tip, while maintaining the orientation of the gripper, will the corresponding entry in the pushability vector be set to true.

Similarly, the graspability affordance on a certain pose is determined by computing a path from the current pose to one located a small distance up. If such a path exists, the graspability on that pose is set to true.

In the end, a pushability map, $Pushability(x, y, z, gripper\_orientation)$ and a graspability map $Graspability(x, y, z, gripper\_orientation)$ will be associated to the table and the gripper in use.

### B. Object

*1) Object Classification:* In a real world environment, there is a large amount of objects which can hardly be individually distinguished, mainly due to noise in the perception. In particular, off-the-shelf cameras and depth sensors are not able to discriminate fine details in the object surface. Thus, the possible discrimination ability of the sensors are limited to simple shapes. What is proposed here is a classification based on the dimensions of a bounding box associated to a certain object.

The classification is given according to the size of each dimension of the bounding box. The reference frame associated to the object is composed of axis $X_{Ob_0}$, $Y_{Ob_0}$ and $Z_{Ob_0}$. To the largest dimension of the bounding box is assigned the $Z$-axis, to the second largest the $Y$-axis and finally to the smallest dimension the $X$-axis.

Having assigned the axis to the right dimensions, the object is classified based on the dimensions' sizes. A constant $D$ is defined to be equal to 3 cm and the classification is done according to the rules on table II-B1.

| # | Class | $X_{Ob}$ | $Y_{Ob}$ | $Z_{Ob}$ | $\#Parts$ |
|---|---|---|---|---|---|
| 1 | SSS | $0 < x \leq D$ | $0 < y \leq D$ | $0 < z \leq D$ | 6 |
| 2 | SSM | $0 < x \leq D$ | $0 < y \leq D$ | $D < z \leq 2D$ | 10 |
| 3 | SSL | $0 < x \leq D$ | $0 < y \leq D$ | $z > 2D$ | 14 |
| 4 | SMM | $0 < x \leq D$ | $D < y \leq 2D$ | $D < z \leq 2D$ | 16 |
| 5 | SML | $0 < x \leq D$ | $D < y \leq 2D$ | $z > 2D$ | 22 |
| 6 | MMM | $D < x \leq 2D$ | $D < y \leq 2D$ | $D < z \leq 2D$ | 24 |
| 7 | SLL | $0 < x \leq D$ | $y > 2D$ | $z > 2D$ | 30 |
| 8 | MML | $D < x \leq 2D$ | $D < y \leq 2D$ | $z > 2D$ | 32 |
| 9 | MLL | $D < x \leq 2D$ | $y > 2D$ | $z > 2D$ | 42 |
| 10 | LLL | $x > 2D$ | $y > 2D$ | $z > 2D$ | 54 |

TABLE I
OBJECT CLASSIFICATION.

| Rotation Index, $i$ | $\alpha$ | $\beta$ | $\gamma$ | Reference Axis |
|---|---|---|---|---|
| $i = 0, ..., 11$ | $0°$ | $0°$ | $30°i$ | $Z$ up |
| $i = 12, ..., 23$ | $90°$ | $0°$ | $30°(i - 12)$ | $Y$ up |
| $i = 24, ..., 35$ | $0°$ | $270°$ | $30°(i - 24)$ | $X$ up |
| $i = 36, ..., 47$ | $180°$ | $0°$ | $30°(i - 36)$ | $Z$ down |
| $i = 48, ..., 59$ | $270°$ | $0°$ | $30°(i - 48)$ | $Y$ down |
| $i = 60, ..., 71$ | $0°$ | $90°$ | $30°(i - 60)$ | $X$ down |

TABLE II
POSSIBLE OBJECT ORIENTATIONS.

*2) Object Parts:* After being classified, the objects can be divided into parts. Each facet of the bounding box is divided into parts according to the lengths of its sides. Using the same discretization constant $D$ used previously in the object classification, each dimension is divided into 1, 2 or 3 parts, if they are, respectively, less or equal to $D$, to $2D$ or larger than $2D$.

The ordering of the object parts depends on the order of the facets. Here, the order of the facets is: Front (X+ -axis), Right (Y+ -axis), Top (Z+ -axis), Back (X- -axis), Left (Y- -axis), Bottom (Z- -axis). The parts in each facet are ordered, as if the facet was being looked at directly in front, from left to right and from top to bottom.

*3) Object Local Representation:* Generally, regardless of the object class or number of parts, the objects' orientation is discretized into 72 possible rotations with respect to the object reference frame $(X_{Ob_0}, Y_{Ob_0}, Z_{Ob_0})$, whose origin is located at the center of the bounding box. These are computed according to the Fixed-XYZ axis convention, but any other convention might have been chosen. Rotations around X or Y can only be multiples of $90^o$, as a box cannot stay on an unstable pose. Rotations around the Z-axis are discretized into bins of $30^o$.

Note that, for the sake of simplicity, the orientation of the object reference frame should be the same as the the world frame's, that is

$$ {}^{W_F}_{Ob_0}T = \left[\begin{array}{c|c} \mathrm{I} & {}^{W_F}_{Ob_0}P \\ \hline 0 & 1 \end{array}\right] \quad (1) $$

*4) Object's State Transition Models:* An object's state consists of an orientation, a position and a part which is of interest to perform a given action. The object's orientation and parts are defined as previously exposed.

The position of the object is defined in a local 3x3 grid, with the object at the centre of it. Initially, before any action is performed, the object is at the centre position. However,

after a push action, the object's position may be at any of the 9 discrete locations defined by the grid. This is illustrated in figure 1, where an object is placed in the middle cell (number 4) of the grid and after some push action, it is moved to cell 2, with a different orientation. Then, when the planning is considered on the new object's pose, the grid is updated such that the object is located at its centre, with the correct orientation. Notice how the grid's orientation respects the object's new orientation. The canonical grid orientation is rotated of $\gamma$ around the object's fixed axis $Z^{Ob_0}$.

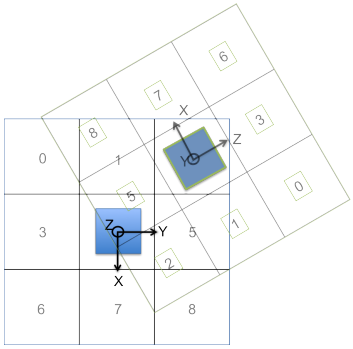



Fig. 1. Object Local Representation for Push Transition Model.

In summary, the full state of an object can be represented by a set of states:

- $O_t = \{1, ..., 10\}$, object class;
- $O_p$, object part, whose domain depends on the object class (see table II-B1);
- $Pos = \{0, ..., 8\}$, object's position on the grid;
- $Ori = \{0, ..., 71\}$, object's orientation

Using this defined state, two local transition models for each object class, $obj \in O$, are defined to represent the result of each of the action types defined, which are push or grasp actions.

The full transition model for the push action of an object, ${}^{obj}T_{push}(Ori, O_p) = P(Pos', Ori'|Ori, O_p)$ is composed of an initial object orientation and an object part to be pushed, which results in some grid position and object orientation, following some probability distribution.

The model used for grasping actions, ${}^{obj}T_{grasp}(Ori, O_p) = P(Success|Ori, O_p)$ is a binomial distribution, that is, a grasp of an object at a given orientation and part will either be successful or unsuccessful.

### C. Gripper

*1) Robotic arm and gripper:* To handle the objects, the robotic arm's end effector is a gripper. The robotic arm base frame, $G_0$, can be defined anywhere in the world frame, translated into the transformation matrix ${}^{W_F}_{G_0}T$. To help defining the reference frame for the gripper, a "thumb" and a "tip" are defined in the gripper. The reference frame for the gripper is shown in figure 2. The thumb is placed on the positive X-axis and the arm continuation along the positive Y-axis.

*2) Gripper's Orientation in World Frame, ${}^{W_F}_{G_1}R$:* As it happens for the object, the gripper's orientation in the world frame also has 72 possible orientations, computed using the Fixed-XYZ axis convention, as shown in table II-B3.

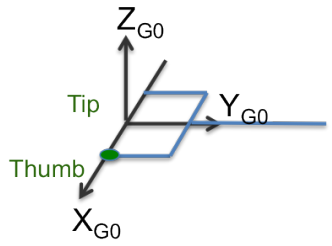



Fig. 2. Reference frame of the gripper.

| Thumb | # | $\alpha$ | $\beta$ | $\gamma$ | Tip Direction |
|---|---|---|---|---|---|
| $X+$ | 0 | 0° | 0° | 0° | $Y-$ |
| | 1 | 90° | 0° | 0° | $Z-$ |
| | 2 | 180° | 0° | 0° | $Y+$ |
| | 3 | 270° | 0° | 0° | $Z+$ |
| $Y+$ | 4 | 90° | 0° | 90° | $Z-$ |
| | 5 | 180° | 0° | 90° | $X-$ |
| | 6 | 270° | 0° | 90° | $Z+$ |
| | 7 | 0° | 0° | 90° | $X+$ |
| $Z+$ | 8 | 0° | 270° | 270° | $X-$ |
| | 9 | 0° | 270° | 0° | $Y-$ |
| | 10 | 0° | 270° | 90° | $X+$ |
| | 11 | 0° | 270° | 180° | $Y+$ |
| $X-$ | 12 | 180° | 0° | 180° | $Y-$ |
| | 13 | 90° | 0° | 180° | $Z-$ |
| | 14 | 0° | 0° | 180° | $Y+$ |
| | 15 | 270° | 0° | 180° | $Z+$ |
| $Y-$ | 16 | 90° | 0° | 270° | $Z-$ |
| | 17 | 0° | 0° | 270° | $X-$ |
| | 18 | 270° | 0° | 270° | $Z+$ |
| | 19 | 180° | 0° | 270° | $X+$ |
| $Z-$ | 20 | 0° | 90° | 270° | $X-$ |
| | 21 | 0° | 90° | 0° | $Y-$ |
| | 22 | 0° | 90° | 90° | $X+$ |
| | 23 | 0° | 90° | 180° | $Y+$ |

TABLE III
GRIPPER'S POSSIBLE ORIENTATIONS WITH RESPECT TO OBJECT CURRENT ORIENTATION.

*3) Gripper's Pose with respect to Object's Frame, ${}^{Ob_1}_{G_1}T$:* For the purpose of grasping and pushing, it is defined a local orientation of the gripper with respect to the object current pose. This local orientation is discretized into 90° rotations around the fixed axes of the object's current frame.

A list of the possible orientations of the gripper with respect to the object current pose, ${}^{Ob_1}_{G_1}R$ is given, in terms of rotation angles $\alpha$, $\beta$ and $\gamma$, in table II-C3.

*4) Transformation from Gripper's frame to World frame, ${}^{W_F}_{G_1}T$, and to Gripper's base frame, ${}^{G_0}_{G_1}T$:* A typical scenario of a simple object manipulation problem is depicted in figure 3. The frames of the various components so far described are included here: the world frame, $W_F$, the gripper's base frame, $G_0$, the object's reference frame, $Ob_0$, the object's current frame, $Ob_1$, and the gripper's current frame, $G_1$.

The transformation from gripper's pose with respect to the current object to its pose in the world frame, ${}^{W_F}_{G_1}T$, is given by (2)

$$
\begin{aligned}
{}^{W_F}_{G_1}T &= {}^{W_F}_{G_0}T.{}^{G_0}_{G_1}T = {}^{W_F}_{Ob_0}T.{}^{Ob_0}_{G_1}T \\
{}^{G_0}_{G_1}T &= {}^{G_0}_{Ob_0}T.{}^{Ob_0}_{Ob_1}T.{}^{Ob_1}_{G_1}T
\end{aligned}
\tag{2}
$$

The transformation from the end-effector pose to the current object position, ${}^{Ob_1}_{G_1}T$, is given by a rotation matrix, ${}^{Ob_1}_{G_1}R$ and a position vector, ${}^{Ob_1}_{G_1}P$. The possible orientations are discretized

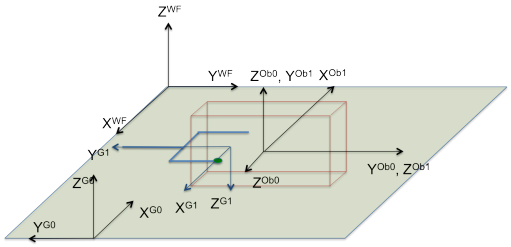



Fig. 3. Frame representation of the various components of an object manipulation scenario; $W_F$ - World Frame; $G_0$ - Gripper's base frame; $Ob_0$ - Object's reference frame; $Ob_1$ - Object's current frame; $G_1$ - Gripper's current frame.

into 24 set of rotation angles, as seen before. The position of the gripper depends on which part of the object is intended and which kind of action is to be performed on it. If it is a push action, the object part gives the desired position for the tip of the gripper, whereas if it is a grasp, the object part determines the position of the thumb. Nonetheless, for each object part, there are only 4 possible orientations for the end-effector.

As described in the previous section, the object's orientation is also discretized into one out of 72 possible orientations. The transformation ${}^{Ob_0}_{Ob_1}T$ is represented only by this possible rotations, since the position vector is always 0, which means that the reference frame always moves along with the object. This makes sense, since the object uses its own local representation.

Having this information, it is possible to compute the transformation from the object reference frame and the gripper's pose, ${}^{Ob_0}_{G_1}T$, following (3).

$$\begin{aligned} {}^{Ob_0}_{G_1}T &= {}^{Ob_0}_{Ob_1}T \cdot {}^{Ob_1}_{G_1}T = \left[\begin{array}{c|c} {}^{Ob_0}_{Ob_1}R & 0 \\ \hline 0 & 1 \end{array}\right] \cdot \left[\begin{array}{c|c} {}^{Ob_1}_{G_1}R & {}^{Ob_1}_{G_1}P \\ \hline 0 & 1 \end{array}\right] = \\ &= \left[\begin{array}{c|c} {}^{Ob_0}_{Ob_1}R \cdot {}^{Ob_1}_{G_1}R & {}^{Ob_0}_{Ob_1}R \cdot {}^{Ob_1}_{G_1}P \\ \hline 0 & 1 \end{array}\right] = \left[\begin{array}{c|c} {}^{Ob_0}_{G_1}R & {}^{Ob_0}_{G_1}P \\ \hline 0 & 1 \end{array}\right] \end{aligned} \tag{3}$$

It is straightforward to compute ${}^{Ob_0}_{G_1}P$, by computing the rotation matrix ${}^{Ob_0}_{Ob_1}R$ using the angles corresponding to the discrete orientation of the object. The rotation matrix ${}^{Ob_0}_{G_1}R$, however, is not directly computed. Instead of computing the two rotation matrices involved from the discrete orientations, a mapping is created from the current orientation of object, $Ob_1$ and the gripper's orientation $G_1$, with respect to the object to an orientation in the object's reference frame, $Ob_0$, as described in (4).

$$\underbrace{{}^{Ob_0}_{Ob_1}Ori}_{72} \times \underbrace{{}^{Ob_1}_{G_1}Ori}_{24} \longmapsto \underbrace{{}^{Ob_0}_{G_1}Ori}_{72} \tag{4}$$

From (2) and (3), one can obtain ${}^{W_f}_{G_1}T$, as seen in (5). Recall that the world frame and the object reference frame have the same orientation, meaning, ${}^{W_F}_{Ob_0}R = I$, which implies that the transformation is a simple translation.

$${}^{W_F}_{G_1}T = \left[\begin{array}{c|c} {}^{Ob_0}_{G_1}R & {}^{Ob_0}_{G_1}P + {}^{W_F}_{Ob_0}P \\ \hline 0 & 1 \end{array}\right] \tag{5}$$

In order to get ${}^{G_0}_{G_1}T$, another transformation is applied to ${}^{Ob_0}_{G_1}T$, which is described in (6).

$${}^{G_0}_{G_1}T = \left[\begin{array}{c|c} {}^{G_0}_{Ob_0}R \cdot {}^{Ob_0}_{G_1}R & {}^{G_0}_{Ob_0}R \cdot {}^{Ob_0}_{G_1}P + {}^{G_0}_{Ob_0}P \\ \hline 0 & 1 \end{array}\right] \tag{6}$$

At this point, ${}^{Ob_0}_{G_1}R$ is computed from the discrete orientation. ${}^{G_0}_{Ob_0}R$ is always known and constant, therefore the rotation matrix of this transformation can be computed. For the translation part of the transformation, ${}^{G_0}_{Ob_0}P$ needs to be updated, since $Ob_0$ moves alongside with the object. The update is done as described in (7).

$${}^{G_0}_{Ob_0}P = {}^{W_F}_{G_0}R^{\top}({}^{W_F}_{Ob_0}P - {}^{W_F}_{G_0}P) \tag{7}$$

The remaining matrices are already known, thus the transformation can be computed.

## III. Learning

### A. Initializations

*1) Graspability and Pushability maps:* The Graspability and Pushability maps are initialized offline, prior to the planning stage. Using the discretization of the table surface, each of the 72 predefined discrete orientations are then used to compute the Inverse Kinematics on the position corresponding to the cell, determining if this pose is reachable.

When building the Pushability map, it is also necessary to verify it is possible to perform a push at that pose. For that, a pose a fixed distance away along the gripper's tip direction is computed, using the same orientation as the preceding pose, and a cartesian path is determined between these two poses. If a smooth path is computable, that is, without great changes in the IK solutions, then the entry on the pushability map corresponding to the first pose is set to *true*, and to *false* otherwise.

A similar situation happens with the Graspability map, except that the following pose is determined along the vertical axis in the world coordinates. Again, if a smooth path is possible between the two poses, the corresponding Graspability map is set to *true*, otherwise it is set to *false*.

*2) Grasp Transition Model:* The grasp transition model ${}^{obj}T_{grasp}(Ori, O_p)$ is initialized as a Bernoulli experiment. For each object type, $obj$, ${}^{obj}T_{grasp}(Ori, O_p)$ is learnt by trying a different grasp orientations on a given object orientation, $Ori$, at a specific object part, $O_p$, and then lift the object to a fixed, small height, while maintaining the gripper orientation. If the gripper can lift the object and it does not fall or change orientation, it is accounted as a successful grasp. Otherwise, the grasp is unsuccessful. For each $(Ori, O_p)$ pair, a fixed number of trials $L_G$ is taken, which is used to compute the probability of success and failure.

*3) Push Transition Model:* Similarly to the learning of the grasp transition model, the push transition model ${}^{obj}T_{push}(Ori, O_p)$ tries different push orientations on the object, $obj$, with a certain orientation, $Ori$, at a given object part $O_p$. The resulting orientation and position are discretized into one of the possible object orientations and position on the object's grid, respectively. Again, for each $(Ori, O_p)$ pair, a fixed number of trials $L_P$ is taken, thus resulting in a categorical distribution for each pair.

### B. Learning from Experience

The system should improve its knowledge of the world as it interacts with the environment. As such, when any action is performed, the execution outcome is used to update the transition models. Depending if the action is a grasp or a push, the number of trials $L_G$ and $L_P$, respectively, are incremented and the row corresponding to the $(Ori, O_p)$ pair in question is updated.

## IV. Planning

### A. Planner

A typical task for a manipulator is to move an object from one place to another without colliding with obstacles. For this robotic manipulator, two type of actions are allowed: pushing and grasping. Given the different nature of these two actions, different approaches are used on how to plan trajectories for each type of action. In the end, a planning algorithm able to accommodate both types of actions is the desired goal. The devised algorithm makes use of a Rapidly-exploring Random Tree approach for planning push actions, due to the highly probabilistic nature of this type of action, whereas for grasping the algorithm is much simpler. Both these algorithms and a way to combine them are explained in the following sections.

*1) Pushing:* The Rapidly-exploring Random Tree (RRT) [5] is a sampling-based motion planning approach, whose main idea is to create a path by randomly choosing poses in the configuration space. The RRT allows planning in environments with many obstacles without the need to explicitly represent them. It also allows for an exploration of the complete state space by sampling. The RRT is used here as a trajectory planner for the object to achieve its desired configuration, that is, a given final position on the table and orientation, when using push-type actions. It is particularly useful for these kind of path planning problems, since the state space is very large and the push actions are highly unpredictable, which would make a MDP algorithm unviable in terms of computer memory usage. The basic RRT algorithm, adapted to include a push planner, is outlined in algorithm 1.

**Algorithm 1** Basic RRT

```
nearNode ← initialNode
while distance(nearNode, Goal) > Threshold do
    target ← ChooseTarget(Goal)
    nearNode ← Nearest(target)
    newNode, actionSet ← LocalPushPlanner(nearNode, target)
    AddNode(newNode, nearNode, actionSet)
    nearNode ← newNode
end while
```

**Algorithm 2** ChooseTarget(Goal)

```
p := constant
0 ≤ r ≤ 1
if r ≤ p then
    return Goal
else
    return RandomState()
end if
```

**Algorithm 3** Nearest(target)

```
for each node in rrt do
    dist ← Distance(node, target)
    if dist < bestDist then
        bestDist ← dist
        nearestNode ← node
    end if
end for
return nearestNode
```

Given the initial and final object's poses, the RRT algorithm explores the configuration space in order the get a path between these two poses. The root of the tree is the initial pose, which is expanded repeatedly until one of the leaves satisfies a distance criteria to the goal pose. The distance criteria should depend on the task the planner was set to.

First, a new state is chosen from the state space; in this case, an object position on the table and its orientation. The process of choosing the next target is outlined in algorithm 2. With probability $p$, the actual goal state is chosen as the node towards which the tree should be extended and with probability $1 - p$ a random configuration is chosen. Then, the tree is searched for the node which is closest to the target node selected on the previous step, which is represented by the $Nearest()$ function. Having chosen the nearest node in the tree to the target, the tree is extended by the $LocalPushPlanner()$ algorithm, outlined in Algorithm IV-A1 and explained further ahead. Finally, after extending the tree towards the target node, the resulting node from the extension, that is, the local push planner, will be added to the RRT. When a new node is added, it registers which node is its parent. Having updated the RRT, the loop continues until the stopping criteria is met.

**Algorithm 4** LocalPushPlanner(nearNode, target)

```
N, MaxIter, MaxFail, eps
bestDist ← distance(nearNode, target)
currentNode ← nearNode
while newDist > eps and nIter < MaxIter and nFail < MaxFail do
    actionSet ← getNRandomPushes(N)
    newState, newDist, actionSelected ← selectAction(actionSet, currentNode, target, bestDist)
    if newState == Empty then
        nFail++
    else
        actionList.AppendActionToList(actionSelected)
        nIter++
        currentState ← newState
    end if
end while
return currentState, actionList
```

The $LocalPushPlanner()$ function extends the tree towards the desired target node. Since each push action will not move an object very far, saving the resulting states for each push and adding them to the RRT would result in a very large tree with nodes not very differentiated. Therefore, instead of saving a new node per push action performed, a greater number of actions is first performed and only the state resulting from that series of pushes is added to the tree. It is defined

a maximum number of pushes performed, $MaxIter$, before adding the node to the RRT. Also, only a maximum number of failed actions, $MaxFail$, is allowed. The meaning of a failed action in this context will be clear when the $selectAction()$ function is explained.

The $LocalPushPlanner()$ function starts by getting a possible action set, by sampling randomly $N$ push actions ($getNRandomPushes()$). A push action corresponds to selecting an object part that is possible to push when the object is at a given orientation, that is, is not on the facet against the table. Then, from this set of actions, one of them is selected according to $selectAction()$ and the resulting state and its distance to the target are computed. If no action is selected, the failed actions' counter, $nFail$ is incremented. Otherwise, the chosen action is added to a list of actions and the counter of the number of pushes performed, $nIter$, is incremented. This process is repeated until the distance between the resulting node and the target is less than a threshold, $\epsilon$, or the number of failed actions or pushes performed reaches a threshold value. The resulting state and list of actions are kept together, so that the information on how that state was reached is saved.

**Algorithm 5** selectAction(actionSet, currentNode, target, bestDist)

```
while actionSet is not empty do
    rAction ← actionSet.pop()
    objPart, objOri ← actionInfo(rAction)
    if computePushability(currentNode, objOri, objPart)
    == false then
        continue
    end if
    newState, actionCost ← applyAction(rAction)
    dist ← distance(newState, target)
    if dist < bestDist then
        bestDist ← dist
        bestAction ← rAction
        resState ← newState
        bestCost ← actionCost
    end if
end while
updateRRTcost(bestCost)
return resState, actionList
```

Going back to the $selectAction()$ function, it is necessary to recall that the object has its own representation of the transitions that occur from a push action, ${}^{obj}T_{push}(Ori, O_p)$, depending on its orientation and pushed object part. The actions from the randomly generated action set have information on the orientation of the object and which part is being pushed. It is necessary to determine the coordinates of the object part in the world frame and check if it is possible to push it, by consulting the pushability map $Pushability(x, y, z, gripper\_orientation)$. In case it is not, that action is discarded and the next one in the set is tested. If the action set is empty and no action was possible, the function will return an empty state, meaning it has failed.

The object's transition table associated to the push action, ${}^{obj}T_{push}(Ori, O_p)$, has different probabilities assigned to each of the possible resulting states. When applying an action, ${}^{obj}T_{push}(Ori, O_p)$ is used to weigh a random sampling of the resulting state. This random sampling makes will still most likely pick the state with highest probability, while allowing for some exploration, important to the continuously learning process and also to avoid getting stuck in badly learnt resulting states. The applied action will have a cost associated to it, which can be computed as $1 - pr$, where $pr$ is the probability of that action actually resulting on the previously mentioned state.

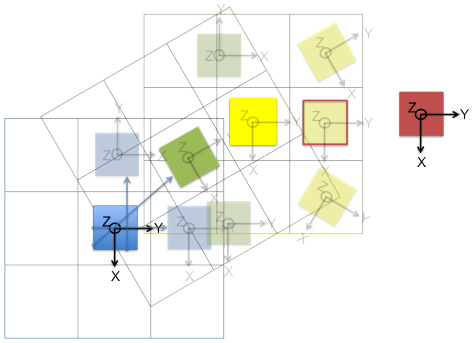


Fig. 4. Local Push Planner. The initial and final configuration of the object is represented by the bold blue and red squares, respectively.

The local representation of the resulting state consists on a discrete orientation and a position in a $3 \times 3$ grid, which needs to be translated to a global representation of the object's position and orientation. The global position and orientation of the object is computed in this function, to compute the distance from the resulting state and the goal state. Having run all the actions in the action set and computing this distance for each of the resulting states, the distances are compared against each other. The action to be selected is the one corresponding to the smallest distance, unless this distance is greater than the one achieved in one of the previous iterations of the $LocalPushPlanner()$, in which case it will return an empty state and be recorded as a failing action.

An illustration of the *LocalPushPlanner* is depicted in figure 4, using a maximum of 3 iterations when extending the tree. The initial object pose is represented by the bold blue block and the goal pose by the red on. The arrows represent some of the random actions taken at different parts of the object, pointing at the expected outcome of such push action, represented by the shaded boxes. Computing the distance to the goal, the planner selects the one pose that minimizes it, represented here by the full green box. This node is selected to proceed in the planning for the second iteration. The grid moves along with the object, as described before, and a new set of actions is taken, resulting in new possible states. The one in yellow is chosen and goes through the same process. The third and last iteration determines that the final node is the one with the red boundaries and this is the node to be added to the RRT.

After building the RRT, there will be a path from the initial pose to the desired goal, with some error allowed for the final pose. The path can be rebuilt starting from the goal node and going up the tree, since each node is aware of its parent and has the set of push actions that led to its corresponding state.

*2) Grasping:* The planning part that corresponds to grasping is much less complex than the pushing part. The outcome

of a grasping action can be more easily modeled. If one considers that after an object being grasped, it will not slip or fall from the gripper, it is possible to take it to any other place on the table, as long as it is within reach, thus becoming a geometric problem only. With this assumption, the grasping action result can be divided into successful or unsuccessful.

A grasp corresponds to placing the gripper's thumb in contact with a part of the object with a given orientation and then lift the object vertically. If the object does not fall or slip, the grasp is considered successful and is otherwise considered unsuccessful. As it can be seen, this can be modeled as a binomial distribution, assigning a probability to each case.

When planning to perform a grasp, it is necessary to take several aspects into account. Algorithm 6 summarizes the planning taken in grasping.

**Algorithm 6** Grasp Planning

```
Initial Conditions:= Object_Ori, Object_Pos
bestGraspSuccess ← 0
bestGripperPose ← NULL
dim:=0, face:=0
while dim < 3 and face < 6 do
  for Obj_Part in parts in face do
    possibleGripperOrientation = {}
    for gripper_ori = 1 : 4 do
      if objectPartGrippable(gripper_ori, dim, Object_Part, Object_Ori) then
        possibleGripperOrientation.push(gripper_ori);
      end if
    end for
    if possibleGripperOrientation not Empty then
      Gripper. computeGripperFrameOriginWRTObject(Obj_Part)
      for gripperOri in possibleGripperOrientation do
        Gripper.updateGripperOrientation(Object_Ori)
        if getGraspability(Gripper.Pose) then
          graspSuccess ← obj T_grasp(Object_Ori, Obj_Part)
          if graspSuccess > bestGraspSuccess then
            bestGraspSuccess ← graspSuccess
            bestGripperPose ← Gripper.Pose
          end if
        end if
      end for
    end if
  end for
end while
```

Before running the algorithm for grasping, it is determined if the object is possible to grasp at a certain orientation. If so, the planner verifies if the object is in a position within grasp's reach. Only if these two conditions are met will the planner proceed with the grasping algorithm.

For each object part, there are four possible orientations for the gripper with respect to the object, since the thumb position is defined over that part. Hence, it is necessary to assert if any of these orientations can be used to grasp the object, which is related to the length of the "fingers" of the gripper. Afterwards, it is important to check whether some part of the object is graspable at a given position on the table. Using only the possible gripper orientations for the object part being evaluated, the gripper's pose is computed.

Having computed the gripper's pose in the world frame, the environment's graspability map, $Graspability(x, y, z, gripper\_orientation)$ is used to determine whether that pose is achievable or not. In case it is achievable, the cost of that grasp is computed by consulting the probability of success in grasping that object's part in the given orientation, which is given by the transition model ${}^{obj}T_{grasp}(Object\_Ori, Obj\_Part)$. In the end, the grasp that leads to the greatest probability of a successful grasp is used.

*3) Mixing Grasping and Pushing in Planning:* The planner makes use of both pushing and grasping actions to plan a path towards the goal pose of the object. In some cases, the object cannot be grasped, but a push action may be possible, as long as the object is within a pushable region. Therefore, two paths are computed. One of them is a push-only path, where only push actions are used. The second one uses grasp actions and may also include push actions when the object is out of grasp. The push-only path is a computed using a regular RRT approach, as described before. In the end, a path from the initial pose to a final pose is obtained, along with the cost associated to it. The second path can be a mixture of push and grasp actions or just a simple grasp.

First, the graspability at the initial pose of the object is evaluated. If the object can be grasped, then it will be grasped. Otherwise, the planning algorithm tries to compute a path using pushes to a point where a grasp can be performed. For this purpose, a set of randomly chosen points in the neighborhood of the initial pose is determined and an RRT is computed from the initial point to each one of the poses contained in this set. For each pose reached in which a grasp can be performed, there is a corresponding RRT cost, which is then associated with the grasping cost in that pose. The RRT and grasping pose chosen is the one that leads to the smallest overall cost.

With the grasp performed, the planner will try to take the object to the goal position. If, however, the final pose cannot be reached directly from a grasp, the object is left at a point close to the goal, towards which the gripper will try to perform pushes. The point in which the gripper will release the object and then start performing pushes is determined in a similar way to the one described before. Another set of random poses in the neighborhood of the goal is determined and an RRT is computed from each of these poses to the goal pose. Finally, the pose chosen is the one that correspond to the less costly RRT. The overall cost of this approach is the grasping cost, plus the possible initial and final push trajectories' cost.

In the end, the planner has computed two possible paths and the one with the smallest cost is chosen and performed.

## V. Results

The planning algorithm described in this dissertation was implemented in C++, in order to reduce the computation time taken and to be able to use tools that require this programming language. The Robot Operating System middleware, better known as ROS [6], provides a vast array of tools, including some meant for visualization and planning. The geometric planning tool used was *MoveIt!* [7], a geometric planner able

to generate trajectories in the joint space from the cartesian trajectories, while avoiding collisions with other objects in the scene. It also provides a way to communicate with a physics simulator called Gazebo [8], which can be used to execute the planned trajectory on a robot, though it has not been used for that purpose.

The robot chosen to perform the experiments was the iCub, which has its own physics simulator that communicates over YARP [9] (*Yet Another Robot Platform*). Due to compatibility constraints between ROS and YARP, the grasping part of the algorithm was not tested, even though it was implemented.

### A. Push Transition Model

The object type chosen to illustrate the concept here described was of type SSL, of dimensions 3x3x10 cm, with a discretization of $D = 3cm$, which has 14 parts. The object is placed in Gazebo, the chosen physics simulator for this learning step, due to the extended capabilities of this simulator against the iCub's. To simulate a push action, a sphere of radius $0.5cm$ is chosen as the pushing object. The mass of the sphere is 10 times bigger than the SSL object, as to be able push it.

In order to learn the push transition model of this object class, the object position is set at the origin of the simulator's reference frame. The orientations are chosen according to table II-B3, using the canonical orientation ($\gamma = 0$) for each of the reference axis there described. In total, 6 orientations are selected, one for each reference axis.

For each orientation, the simulator performs pushes on the parts that are neither facing the table nor parallel to it. For each part, the pusher is placed in contact with the object. To introduce some error on the sensing of the object position and on the precision of the manipulator, the pusher is placed on multiple positions around the centre of the part, performing pushes in various direction against the facet. The length of the push is $3cm$ along the specified orientation.

After each push, the resulting pose is extracted, taking the final position and orientation, which is then discretized into one of its possible values, 9 for the position and 72 for the orientation. The process is repeated for each part and for all the canonical orientations.

The push transition model can then be initialized for each of the canonical orientations and the parts. For the remainder of the orientations, the transition models are inferred from their canonical correspondent, rotating the final orientations $\gamma$ around $Z$.

Part of this process is shown in figure 5, in which the SSL object is initially placed in the centre of the grid, with the $Z$-axis pointing down, as in figure 5(a), and then pushed on part 0 (where the white sphere is initially located). The resulting state can differ, as it can be seen in figures 5(b), 5(c) and V-A.

In image 6, a row from the resulting push transition model for the SSL object is represented in the chart. It corresponds to discrete orientation 36 (Z-down), part 0, showing the uncertainty inherent to this push. It will most likely get to position 1 (red column) with orientation 36, with 0.3 probability, but can also reach other poses.

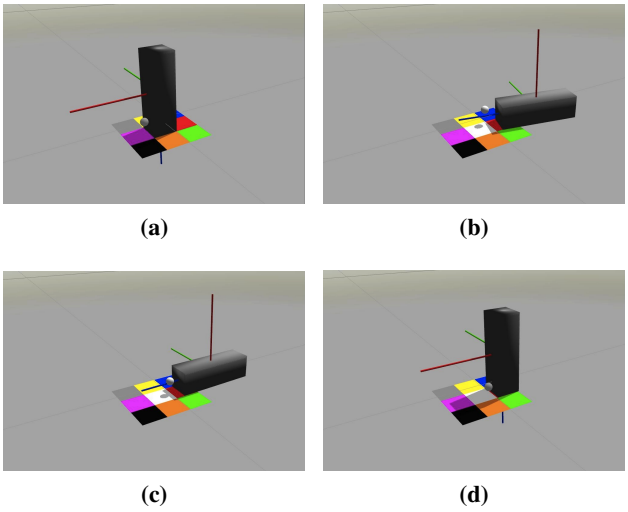



Fig. 5. Push Simulation. The initial pose is shown in (a), where the X, Y and Z-axis are represented by the red, green and blue axis, respectively. The object's grid is represented on the floor by the colored squares. The object is pushed on part 0 by the white sphere and examples of the resulting poses are shown in (b), (c) and (d).

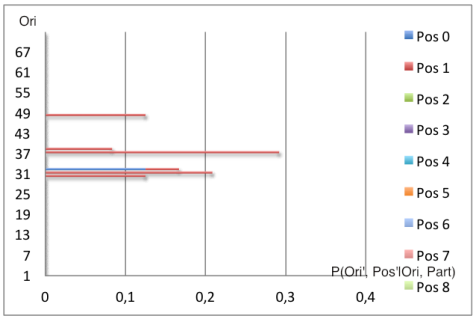



Fig. 6. Push Transition Model for discrete orientation 36 and object part 0.

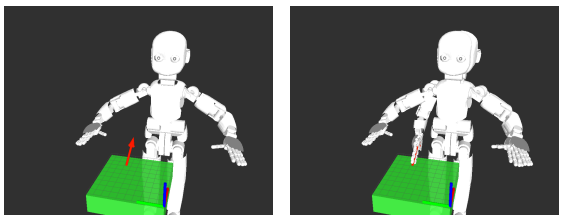
### B. Pushability Map

As described in section III-A1, the pushability map is learned from the geometric model of the robot. The dimensions of the table are 30x30 cm, displaced to the right side of the robot, in order to improve the right hand's reachability. The maximum allowed height for the gripper with respect to the table top was set to 18 cm, thus resulting on a table discretization of 600 cells.

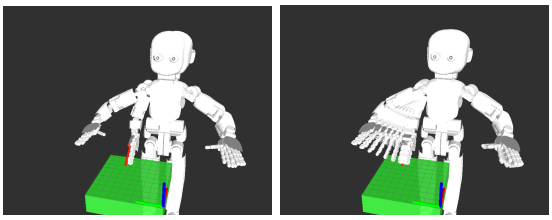
Figure 7 shows a model of the iCub robot in *MoveIt!*, performing a push on a certain position of the table grid. The gripper point chosen to execute the push tries to reach the table position at a given pose, represented by the red arrow (figure 7(a)). When the target pose is reached (figure 7(b)), it tries to reach a pose with the same orientation, but displaced $3cm$ on the direction normal to the palm of the hand (figure 7(c)). The full trajectory taken by the robot is depicted in figure 7(d).

The table, represented by the green object, is used as a collision object, which the trajectory planner uses to avoid collisions between the robot and the table.

The pushability affordance is computed for each cell of the table, at each of the 72 discrete orientations allowed to the gripper. To improve its affordance, for each orientation, a tolerance of $\pm 30^{\circ}$ is given to the gripper's orientation around



**(a)** Target Pose **(b)** First waypoint reached



**(c)** Push Concluded **(d)** Push Trail

Fig. 7. Computing pushability at a certain pose.

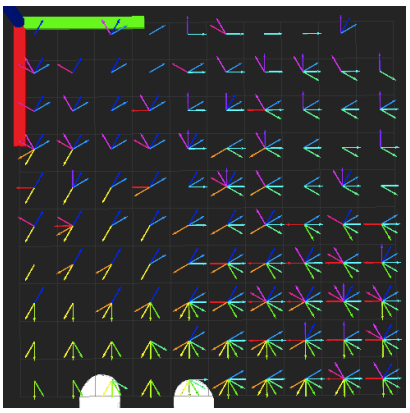


Fig. 8. Representation of the pushability map. Only a maximum of 24 discrete orientations are shown, corresponding orientations in which either the Z-axis is pointing up or down.

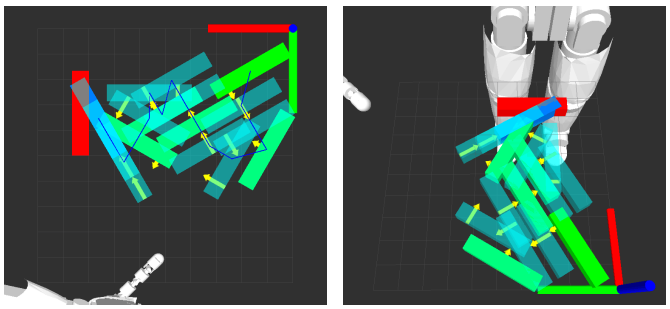
the X-axis on the gripper's schematic representation (in the gripper's simulation frame, around Z-axis).

The resulting pushability map is illustrated in figure 8, showing the pushability affordance on the lower level of cells over the table. The total time taken to initialize the pushability map was about 5 hours and 7 minutes, each cell taking around 30 seconds to test.

### C. Planning Push Actions

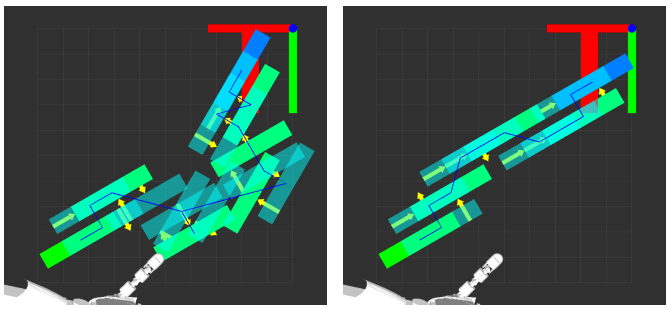
Having initialized the pushability map for this setup and the push transition model for objects of type SSL, the planning stage can take place. In these settings, the distance criteria is the Euclidean distance between two poses, using a *threshold*=3cm as stoping criteria. The probability $p$ of extending the RRT towards the goal is 0.1, and the parameters for the Local Push Planner are $eps = 0.1$, $N = 300$, $MaxFail = MaxIter = 3$.

An example of planned trajectory for the gripper and expected states of the object at each stage is presented in figure 9. The initial object pose, given by the green box, is set at position $(5, 5)cm$ and orientation $(0°, 90°, 30°)$. The



**(a)** Top-down view. **(b)** A different perspective.

Fig. 9. Push Planning Example. The initial pose of the object is represented by the green pose, the target pose by the red one. The solid blue represented the actual final pose achieved by the path planning algorithm, the sea green boxes show the pose corresponding to the RRT nodes and the transparent blue boxes the states the object takes between two nodes. The yellow arrows represent the push direction, length and base point. The blue line in (a) shows the path taken by the object



**(a)** Using iCub's Pushability Map, top-down view. **(b)** Using Ubiquitous Pushability Map, top-down view.

Fig. 10. Comparison of different Pushability Maps. In (a), the iCub's pushability map is used, whereas in (b)an ubiquitous pushabiliy map is used instead.

target pose represented in red is in position $(25, 10)cm$ and orientation $(0°, -90°, 100°)$ . The sea green boxes represent the object pose of a node in the RRT path and the transparent blue boxes represent the transient states of the object between two nodes. The actual final pose reached is given by the solid blue box. The blue line in figure 9(a) shows the planned path taken by the object, starting on the green box and finishing in the solid blue one.

The yellow rows illustrate the push action at each step. The tail is placed on the desired object part to be pushed and the arrow points along the direction of the push, ending in the final gripper pose for this action.

The computed path uses 6 out of the 15 nodes of the RRT, which represents, in total, 15 pushes to achieve the desired goal and, hence, 15 intermediate object poses.

To test the effect of the pushability map on the planning stage, another setup was experimented, setting the initial pose to position $(25, 25)cm$ and orientation $(0°, 90°, 30°)$ and the final pose to position $(5, 5)cm$ and orientation $(0°, -90°, 100°)$.

Using the pushability computed for the iCub, the result is shown in figures 10(a). The total number of nodes in the resulting RRT is 9, although only 7 belong to the path.

If the same setup is used, but using an ubiquitous manipula-

TABLE IV
PUSH PLANNING ALGORITHM PERFORMANCE

| | | Case 1 | Case 2 |
|---|---|---|---|
| Initial Pose | Pos | $(20, 10)cm$ | $(25, 12)cm$ |
| | Ori | $(180°, 0°, 60°)$ | $(90°, 0°, 210°)$ |
| Final Pose | Pos | $(10, 20)$ | $(10, 20)$ |
| | Ori | $(0°, -90°, 0°)$ | $(90°, 0°, 0°)$ |
| #Sims | | 200 | 200 |
| #Successes | | 186 | 159 |
| #Failures | | 14 | 41 |
| #Targets generated when successful | Max | 844 | 972 |
| | Min | 2 | 3 |
| | Median | 11 | 36 |
| #Nodes in RRT generated when successful | Max | 18 | 26 |
| | Min | 3 | 4 |
| | Median | 5 | 10 |
| #Nodes in Path when successful | Max | 7 | 8 |
| | Min | 3 | 3 |
| | Median | 4 | 5 |
| #Nodes in RRT generated when failed | Max | 21 | 24 |
| | Min | 13 | 11 |
| | Median | 15 | 17 |

tor, that is, one which could perform a push at any given pose, which corresponds to have a pushability map filled with 1's, the result is as presented in figure 10(b). The RRT is composed of 5 nodes, all of them included in the path, and it is possible to see that the motion is much more linear, straight to the goal, as expected, which shows that robots with different pushability maps would perform differently.

To illustrate the performance of this algorithm, 3 aspects were taken into consideration: number of target nodes generated by the RRT before the Local Push Planner is called, the total number of nodes in the RRT and the length of the path generated. For that purpose, the algorithm was run 200 times on a set of 2 pairs of initial and goal poses for the object, recording the number of failures and successes achieved. It was considered a failure if after generating 1000 target nodes, there was no path found between the two poses.

The result of this experiment is shown in table IV.

The first pair of poses had a success rate of $93\%$, whereas the second pair had only $79.5\%$ of successful plans. The fact that the pushability on the initial pose of the second pair is low, meaning, few push poses are allowed on that position, is the reason why the the success rate is lower.

When the planning is successful, the number of targets generated when building the RRT varies in a wide range, but $50\%$ of the times, it is lower than 11 targets for the first pair and 36 for the second. For the first case, the number of nodes in the RRT varies between 3 and 18, though the median is of only 5. In the second scenario, it can generate from 4 to 26 nodes, with a median value of 10, higher than in the previous case. The statistics in the number of nodes in the path does not differ much when comparing both cases.

The failing criteria used determines that the maximum number of targets created is 1000. The number of targets towards which the RRT could be extended successfully does not present a great difference between the two cases, but the maximum, minimum and median number of nodes in the RRT are very high. In such a simple scenario, such number of nodes is probably too much, so it is better to replan.

## VI. CONCLUSIONS

The dissertation describes an approach to solve the object manipulation on a table top problem, using push and grasp actions the achieve a given task. It presents a combination of sampling-based motion planning approach such as the RRT with a discretization of the state space in a manner that minimizes the computer memory used and still efficiently solves the problem.

A classification of objects based on their dimensions provides a way to generalize to behaviour of similar objects, encapsulated in the push and grasping transition models. The efficient discrete representation of the object's pose and parts provides a suitable way to incorporate the uncertainty of the actions into the planner.

The pushing part of algorithm, based on a RRT approach, was proved to converged to a possible solution most of the times, using a notion of pushability to determine offline which regions of the environment are viable to execute a push, saving a lot of computation time while planning the trajectory to execute.

Since there is not much work done in planning push actions, there is not much background work to compare to the one here developed. One upside of this approach is that it does not involve the use of a physics simulator in the planning process, which would require the knowledge of the exact physics of the objects manipulated. Furthermore, it models the fact that objects may fall during the interaction, which can be easily used in the planning stage to determine to best way to achieve a task.

Future work will involve trying the algorithm in simulation, replanning trajectories as the object moves away from the expected poses, testing the inclusion of grasping in the planner, introducing obstacles in the state space, which can be easily introduced into the algorithm used, and testing with a wider variety of objects.

## REFERENCES


[1] T. Mericli, M. Veloso, and H. Akin, "Achievable push-manipulation for complex passive mobile objects using past experience," in *12th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2013)*, 2013, pp. 71–78. [Online]. Available: http://dl.acm.org/citation.cfm?id=2484935

[2] C. Zito, R. Stolkin, M. Kopicki, and J. Wyatt, "Two-level rrt planning for robotic push manipulation," in *Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on*, 2012, pp. 678–685.

[3] N. Vahrenkamp, T. Asfour, and R. Dillmann, "Robot Placement based on Reachability Inversion," 2013.

[4] G. Metta, L. Natale, F. Nori, and G. Sandini, "The icub project: An open source platform for research in embodied cognition," in *Advanced Robotics and its Social Impacts (ARSO), 2011 IEEE Workshop on*, 2011, pp. 24–26.

[5] S. M. LaValle, *Planning Algorithms*. Cambridge: Cambridge University Press, 2006. [Online]. Available: http://ebooks.cambridge.org/ref/id/CBO9780511546877

[6] http://wiki.ros.org/Documentation, July 2013.

[7] http://http://moveit.ros.org/wiki/MoveIt!, June 2013.

[8] http://gazebosim.org/, September 2013.

[9] http://wiki.icub.org/yarpdoc/index.html, July 2013.