Pushing using Compliance

Dennis Nieuwenhuisen A. Frank van der Stappen Mark H. Overmars
Institute of Information and Computing Sciences
Utrecht University, The Netherlands
Email: {dennis,frankst,markov}@cs.uu.nl

Abstract— **This paper addresses the problem of maneuvering an object by pushing it through an environment with obstacles. Instead of only pushing the object through open spaces, we also allow it to use compliance, e.g. allowing it to slide along obstacle boundaries. The advantage of using compliance is twofold: compliance does not only extend the number of situations in which a push plan can be found, it also allows for simpler (i.e. less complicated) paths in many cases. Here, we present an approach based on the Rapidly-exploring Random Tree (RRT) algorithm that, besides paths through the open space, exploits the power of compliance.**

I. Introduction

Manipulation and motion are common ways of navigating through and interacting with a real or virtual world. In general, motion planning deals with autonomously moving objects, whereas manipulation planning considers passive objects that can only be moved by another entity, the manipulator.

Objects can be manipulated in numerous ways, and each type of manipulation implies different constraints on the combined motion of the manipulator and the manipulated object. In industrial settings for example, many machines require objects to be input in a fixed orientation. Another example is when a large part needs to be removed from a factory for maintenance. This type of problem also finds application in virtual training software (e.g. training firefighters) in which obstacles need to be moved in order to reach a certain area.

All types of manipulation have in common that the manipulator needs to apply force on the object. A lot of research has been conducted on how to apply these forces. This research has resulted in a broad range of different forms of manipulation that include grasping [14], squeezing [10], pushing [14], [15], rolling [3] and even throwing an object [16]. Since many parameters are involved, e.g. mass distribution of the object, different types of friction and limitations of the manipulator, manipulation planning is often a difficult problem.

Pushing is one of the most widely studied classes of manipulation. The objective of pushing is to maneuver the object O, incapable of moving by itself, from an initial configuration to a goal configuration by pushing with a pusher P. A push plan for P needs to be created that pushes O to its destination.

Although pushing is conceptually simple, a complicating factor is the lack of space for P to maneuver to a desired position to push O. This can result in a situation where O gets stuck, for example in a corner. Also, if the space for P to maneuver is limited, only a complicated push plan may be able to push O to its goal. These problems can often be overcome by exploiting the boundaries of the environment. The rationale behind this is that if O is pushed while it touches one of the boundaries of the environment, it is forced to follow a path parallel to that boundary. Such a motion is called a *compliant motion*. If O is compliant, there is a range of directions from which it can be pushed that all cause O to follow the same path, in contrast to the non-compliant situation where there is only a limited number of directions to push O in a certain direction. Thus it is more likely that P can push O from an appropriate direction during a compliant motion than during a non-compliant motion (see Fig. 2 for some examples).

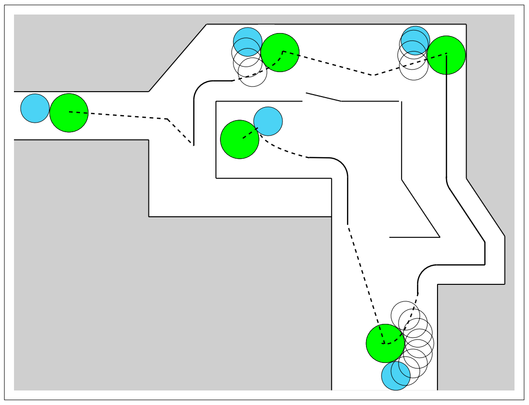
Fig. 1. An example created by our implementation. At the left the start configuration is shown. The object is pushed to the goal in the center using a combination of compliant and non-compliant path segments. Dotted lines represent open space path segments, solid lines represent compliant path segments. In the right top, the start of a compliant path segment is shown, two more examples where the pusher changes its position are also shown.

Although pushing has received considerable attention over the years, using compliance as an aid to maneuver the object with a pusher, has not. Often compliance is used to compensate for uncertainty, for example to solve the peg-in-hole problem. In [13] the *preimage backchaining* approach is introduced. The idea is to compute the points from which the robot can reach the goal. Then the preimage is iteratively treated as a new goal until the initial robot configuration has been found. The concept of a *back projection* is introduced in [9] and [7] improves an algorithm introduced in [8] to find a trajectory from a start region to a goal region amidst planar polygonal obstacles where control is subject to uncertainty in $O(n^2 \log n)$ time. The coordinated motion planning problem of two autonomous robots in a plane using a cell decomposition is solved in $O(n^2)$ time in [19] and [4].

In a previous paper [17] we presented an algorithm that preprocesses an environment consisting of n non-intersecting line segments in $O(n^2 \log n)$ time and is able to create a push plan in $O(kn \log n)$ time where k is the number of segments of the path of O. The algorithm is based on the assumption that the path of O is given. In this paper we lift this restriction by describing an algorithm that is capable of creating a push plan that can contain compliant segments.

Our approach is based on the Rapidly-exploring Random Tree (RRT) [12] algorithm. It combines random exploration of the open space with an exact computation of the compliant space. For the open space, a tree is created by generating random configurations. For every random configuration a path is tried from the tree to that random configuration. If the random configuration cannot be reached because of a collision, a configuration as close as possible to the obstacle that caused the collision is added to the tree. This configuration is then used as a starting point to explore the compliant configurations using an exact computation. The results of this compliant exploration are added as configurations to the tree.

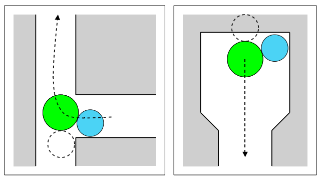

Fig. 2. Two examples in which compliance helps finding a push plan. The path to the goal is shown as the dotted arrow. The dotted positions for the pusher represent the positions that would push the object quickly in the right direction; in both pictures these are not reachable. If no compliance is allowed, in the left picture no path will be found, in the right picture a complicated push plan can be created in which the pusher alternates between two positions. With compliance a simple push plan exists in both situations.

II. Preliminaries

Given disks O and P with radii r_o and $r_p < r_o$ in an environment consisting of disjoint line segments L, and given a start and goal position for O, we compute a push plan for P such that if P follows this plan, it pushes O from its start to its goal configuration. In this paper it is assumed that P always keeps contact with O (see the conclusions for some remarks on this). This allows us to describe the position of P using only an angle. It is assumed that the friction between O and the supporting plane is large enough such that there is no motion of O after pushing ceases (quasi-static assumption). The center of friction of O is the center of the disk.

Suppose O is pushed through the open space by P. A position for P such that O and P touch is called a *push position*. If we need to change the direction of motion of O, P needs to transit to a new push position. Such a transit is called a *contact transit*.

Definition II.1 (contact transit)**.** *A* contact transit *is a motion in which P slides along the stationary O.*

The path of O can consist of both non-compliant and compliant path segments. A compliant position for O is defined as follows.

Definition II.2 (compliant)**.** *A collision-free position p for O, where p represents the position of the center of O is called* compliant *iff $\exists_{l \in L} : d(p, l) = r_o$ where $d(a, b)$ denotes the Euclidean distance between objects a and b and r_o is the radius of O.*

The collection of all compliant positions is called the *compliant space*. If O follows a compliant path around obstacle l, its path can be described by (a part of) the boundary of the Minkowski sum $(l \oplus D)$ where D is a disk of radius r_o. We will refer to this path as the *compliant path* of l. The relative position of O on the compliant path of l can be described by its *compliant position* (Fig. 3a).

Definition II.3 (compliant position on an obstacle)**.** *A* compliant position *$p \in \partial(l \oplus D)$ of O defined on obstacle $l \in L$ describes the relative position of O on the compliant path of l. The set $\partial(l \oplus D)$ is parametrized by a continuous function $P_l : [0, 1] \rightarrow \partial(l \oplus D)$, that satisfies $P_l(0) = P_l(1)$.*

At every compliant position, there is a range of push positions that all cause O to follow the same path in the same direction. Such a range is called the *push range*. The size of this range is dependent on the friction between O and the obstacle and the friction between O and P.

Definition II.4 (push range)**.** *At compliant position p on obstacle l the* push range *describes the continuous range of push positions that cause O to follow the same path. Since there are two directions of motion, at every compliant position there are two push ranges:* PR^+ *for the clockwise motion and* PR^- *for the counterclockwise motion around l.*

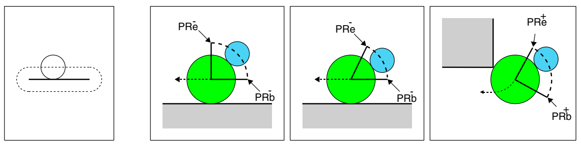

Fig. 3. (a) The compliant positions (dotted line) of an obstacle (the solid line), given an object (disk). On the right, the push range. The dotted arrows show the direction of motion. (b) The push range if no friction is assumed between the object and the obstacle. (c) The push range if there is friction between O and the obstacle. (d) The push range if O is pushed around an endpoint of an obstacle (no friction).

Since the push range is always equal to or smaller than 90 degrees wide, we can define the two extreme values of PR^+ as starting at PR_b^+ and ending at PR_e^+ using the shortest counterclockwise rotation. The same holds for PR^-. Using a contact transit, it is not guaranteed that P can reach either $\text{PR}_b^+/\text{PR}_b^-$ or $\text{PR}_e^+/\text{PR}_e^-$ because one or more obstacles can impede the transit. Examples of the push range are shown in Fig. 3.

A compliant position for O and an angle for P together form a *compliant configuration*. Given a compliant configuration, we define the *bounding obstacles*.

Definition II.5 (Bounding obstacles)**.** *At compliant configuration c and given a direction of motion, the* bounding obstacles *are the obstacles that P hits first if it rotates from its current position to either* $\text{PR}_b^+/\text{PR}_b^-$ *or* $\text{PR}_e^+/\text{PR}_e^-$.

To verify if a compliant configuration has been visited before by the algorithm, it is not enough to simply remember between which compliant positions an obstacle has been visited. As can be seen from Fig. 4, for one compliant position, multiple sets of bounding obstacles can be defined. Between these sets, no contact transit may exist.

Specifying the bounding obstacles for a configuration also specifies the direction of motion, i.e. the bounding obstacles can never be the same for both a clockwise and counterclockwise motion. A continuous part of a compliant path in which the bounding obstacles are fixed is called a *compliant interval*.

Definition II.6 (Compliant interval)**.** *A compliant interval* $(c_b, c_e) : c_b \in P_l \wedge c_e \in P_l$ *defines a continuous set of compliant configurations of obstacle* l *in which the bounding obstacles are fixed.*

A compliant interval is unique, i.e. an interval with the same bounding obstacles cannot occur twice on a compliant path of the same obstacle. Exceptions are the compliant positions where no obstacles bound PR$^+$/PR$^-$(as shown in Fig. 4a). But since these cannot overlap, they can still be uniquely defined using the above definition. Because of the properties of an interval, if a compliant configuration c belongs to an interval, a push plan is guaranteed to exist that pushes O from c to the end position c_e of the interval.

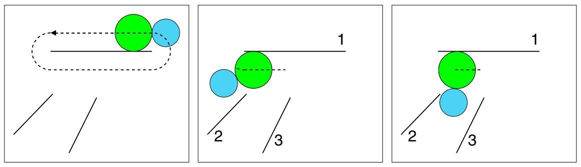

Fig. 4. (a) Starting at the compliant configuration in the first picture (counterclockwise motion), O can reach all compliant positions (dotted line) of the obstacle. (b), (c) Even though O is at the same compliant position in both figures, the bounding obstacles are $(\perp, 2)$ and $(2, 3)$ respectively (where $\perp$ means no bounding obstacle). The dotted lines show the intervals.

Since our final path will consist of both compliant and non-compliant sections, we need a way to leave a compliant path to connect a compliant configuration to a non-compliant configuration. To leave a compliant path efficiently (i.e. limiting the length of the path), a contact transit is used to revolve P to a position as close to the obstacle as possible. If P starts pushing from this position, it pushes O away from the obstacle. If no other obstacles impede the path, the resulting motion of the object is a curve called a *hockey stick curve* (see Fig. 5a). A hockey stick curve minimizes the distance needed to push O away from the influence of the obstacle.

In [1] a mathematical description of a hockey stick curve is given for a point pushing a disk. The resulting motion of a pusher of radius r_p pushing an object of radius r_o is equivalent to that of a point pushing a disk of radius $r_p + r_o$ where the point represents the center of P. The coordinates of O as a function of time are shown in Equations 1 and 2.

To push O from a compliant configuration to a non-compliant configuration, the shortest path is to create a hockey stick curve, followed by a straight line push. We will refer to a combination of a hockey stick curve and a straight line as a *hockey stick push*. An example of a hockey stick push is shown in Fig. 5c, α is the angle the line through the centers of O and P makes with obstacle l.

$$x(t) = t + \frac{1 - \tan^2\left(\frac{\alpha}{2}\right) \cdot e^{2t}}{1 + \tan^2\left(\frac{\alpha}{2}\right) \cdot e^{2t}} - \cos\alpha \qquad (1)$$

$$y(t) = \frac{2\tan\left(\frac{\alpha}{2}\right) \cdot e^{t}}{1 + \tan^2\left(\frac{\alpha}{2}\right) \cdot e^{2t}} - \sin\alpha \qquad (2)$$

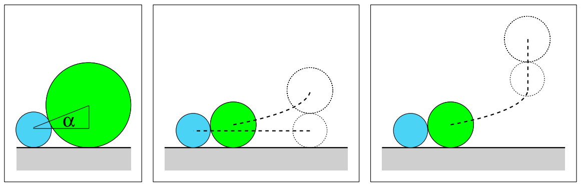

Fig. 5. (a) α is the initial angle between O and P. (b) An example of a hockey stick curve. The dotted lines show the paths of O and P. (c) A hockey stick push. The target of O is shown as the dotted disk. First a hockey stick curve is used, followed by a straight line push. The resulting path is the shortest path to the goal from the start configuration.

III. Rapidly-exploring Random Trees

Many motion planning algorithms are based on the generation of collision-free configurations. Between these configurations connections are tried and a graph is formed that can be used to solve motion planning queries. Since random sampling does not result in compliant configurations, we need a method to create them. Retracting configurations to the obstacles [2] seems a straightforward solution. However this introduces the problem of deciding when and how configurations need to be retracted. Also, retraction of configurations tends to be a costly operation.

After a random configuration has been created, most algorithms try to connect it to already existing configurations in the graph. If the connection fails (because of a collision), an obstacle must be impeding the path. Usually such connections are discarded. In our case however, these collisions are valuable in creating compliant configurations. Since we aim at creating a path from a start to a goal configuration (single shot approach), we use a bidirectional version of the Rapidly-exploring Random Trees (RRT) [12] to create the necessary graph. We will first briefly explain how the RRT algorithm works and then elaborate on how to adapt the RRT such that it is suited to solve our problem.

The RRT is a single shot approach in which a tree is constructed that gradually improves resolution. Here, we will use the bidirectional version of the RRT that grows two exploration trees: T_s from the start configuration c_s and T_g from the goal configuration c_g. As the trees grow larger, the two trees are more likely to connect. If a connection is established, a path is found. The advantage of the bidirectional version over the single directional version is that the first is better at escaping from a local minimum. The algorithm described in this paper is also suited for the approach of using more than two trees, for more information see [11].

The basic RRT algorithm is shown as Algorithm 1. Initially the start configuration c_s is added as a vertex to T_s and the goal configuration c_g is added to T_g. Next, a collision-free random configuration c_r is generated (line 4). The nearest configuration $c_n \in T_s$ to c_r is found (using a Euclidean distance metric) in line 5. Configuration c_n can either be an existing vertex in T_s or it can be a point in the interior of an edge in T_s. In either case, starting at c_n, we move in the direction of c_r. If on the straight line path (c_n, c_r) an obstacle is encountered, the closest configuration c_c to the boundary of the obstacle is determined. Note that we have found a compliant configuration. If c_r is reached without collision then $c_c = c_r$ (line 6). Next, c_c is added to T_s as a vertex.

In order to grow T_g toward T_s, an attempt is made to connect c_c to the nearest configuration c'_n in T_g. Again, if an obstacle is encountered on the path between c'_n and c_c, we connect c'_n to the closest reachable point (c'_c) on the boundary of the obstacle (and we have found another compliant configuration) and add c'_c to T_g as a vertex. If $c'_c = c_c$, the two trees are connected and a solution is found.

Algorithm 1 RRT(S, G, c_s, c_g)

```
 1: $T_s$.AddVertex $(c_s)$
 2: $T_s$.AddVertex $(c_g)$
 3: **repeat**
 4:   $c_r \leftarrow$ random configuration
 5:   $c_n \leftarrow$ GetNearestNeighbor$(T_s, c_r)$
 6:   $c_c \leftarrow$ LocalPlannerForward$(T_s, c_n, c_r)$
 7:   **if** $c_c \neq NULL$ {did we find a legal vertex?} **then**
 8:     $T_s$.AddVertex $(c_c)$
 9:     $T_s$.AddEdge $(c_n, c_c)$
10:     $c'_n \leftarrow$ GetNearestNeighbor$(T_g, c_c)$
11:     $c'_c \leftarrow$ LocalPlannerReverse$(T_g, c'_n, c_c)$
12:     **if** $c'_c \neq NULL$ {did we find a legal vertex?} **then**
13:       $T_g$.AddVertex $(c'_c)$
14:       $T_g$.AddEdge $(c'_n, c'_c)$
15:     **if** PathExists $(c_s, c_g)$ **then**
16:       return FOUND
17: **until** stopping criterion is met
```

Using the RRT as a basic planning algorithm solves the problem of having to decide which and how many configurations to retract to create compliant configurations; the more obstacles are present between c_s and c_g the more compliant configurations will be found and the more likely that the final path will contain compliant segments.

To handle compliance, the RRT algorithm needs to be extended. If the function LocalPlannerForward in line 6 results in a compliant configuration we will need to find out which part of the compliant space is reachable from this configuration, we call this procedure *compliant exploration* (see Section V). Every obstacle has two distinct compliant exploration directions: one in the clockwise and one in the counter clockwise direction.

The object is pushed around the obstacle until it returns

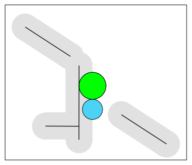
Fig. 6. A scene consisting of 4 obstacles. The Minkowski sums are shown in light gray. The top 3 obstacles form a compliant component while the bottom obstacle is a compliant component on its own (the Minkowski sum does not overlap with any of the others).

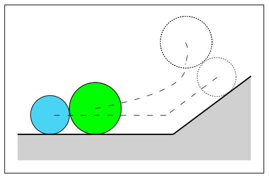
Fig. 7. During the hockey stick push, P encounters another obstacle. From that position, another hockey stick is started etc.

at the start position or until no further pushing is possible. Depending on the circumstances, a compliant path between different obstacles may be possible.

Definition III.1 (compliant component)**.** *A compliant component (Fig. 6) CC, is a maximal group of obstacles that share the following property:*
$\forall_{l \in CC} : \exists_{(l' \neq l) \in CC} : (l \oplus D) \cap (l' \oplus D) \neq \emptyset,$
where D is a disk with the radius of O.

The fact that two obstacles belong to the same compliant component by no means guarantees that a compliant path between them can be created. This also depends on the size and configuration of the obstacles and the size of P.

As pushing motions are irreversible, the connections between vertices are directed. Therefore the paths the local planner creates are only suited for motions in one direction. Swapping the direction of collision checking (e.g. reversing the endpoints) for T_g is no solution because for the proper functioning of the RRT algorithm it is necessary that collision checking starts at the tree and moves in the direction of the random vertex. The solution is to use a “reversed” version of the local planner (line 11), including a reversed version of compliant exploration.

In lines 5 and 10 the nearest configuration in the tree needs to be found. If the nearest configuration (c_n) in the tree is non-compliant, the shortest distance is simply the straight line path. If a compliant configuration is involved, the shortest distance between two configurations may however consist of a (partial) hockey stick curve.

IV. Local Planner

The local planner (lines 6 and 11 of Algorithm 1) is a crucial part of many motion planning algorithms. It connects two configurations to each other, usually by trying a straight line path between them. If the local planner succeeds in finding a path, a connection in the tree between the corresponding vertices is created.

As our configurations do not only consist of a position of O, but also contain a position for P and configurations may or may not be compliant, the local planner needs to adapt its approach according to the circumstances. Also we need two versions of the local planner: a “forward” one (Algorithm 2) and a “reversed” one (Algorithm 3), both will be discussed.

A. Forward local planner

Suppose the nearest neighbor configuration c_n resulting from line 5 of Algorithm 1 is non-compliant (Fig. 8a). The local planner has to verify whether a straight line path exists that connects c_n to the random configuration c_r. There is only one push position for P such that O follows a straight line path in the direction of c_r. To reach this push position, a contact transit is used.

If c_n is compliant to obstacle l (Fig. 8b) it depends on the position of c_r whether P is able to transit to the desired push position. There is a high probability that this will fail because l is likely to impede the contact transit. In that case, we try to push O a distance of $2r_p$ away from l. If this succeeds, we are certain that P now fits between O and l. The most effective way of pushing O away from l is to use a hockey stick push. A hockey stick push maximizes the angle in which P pushes O away from l. Before the end of the hockey stick is reached, P may encounter another obstacle. If this happens, a new hockey stick push is started etc. (Fig. 7). In theory, if O is in a small confined area or if the radii of O and P are (almost) equal, this can continue infinitely. Restricting the total length of the consecutive hockey stick curves solves this issue.

In both situations (c_n being compliant or non-compliant) P has reached the desired push position to try to push O in a straight line from its current position to c_r. If c_r is reached, then a (non-compliant) vertex positioned at c_r is added to T_s together with the edge (c_n, c_r). The position of P of this vertex is set to the position of P at the end of the path. If an obstacle is encountered at configuration c_c before c_r is reached (Fig. 8c), a compliant configuration is found and is used as a starting point for the compliant exploration algorithm of Section V. The complete LOCALPLANNERFORWARD algorithm is shown as Algorithm 2.

Algorithm 2 LOCALPLANNERFORWARD(T_s, c_n, c_r)

```
1:  if COMPLIANT($c_n$) then
2:      $c_n \leftarrow$ CREATEHOCKEYSTICK($c_n$)
3:      if $c_n$ = NULL then
4:          RETURN NULL {failure}
5:  $c_n \leftarrow$ CREATECONTACTTRANSIT($c_n, c_r$)
6:  if $c_n$ = NULL then
7:      RETURN NULL {failure}
8:  $c_n \leftarrow$ PUSH($c_n, c_r$)
9:  if COMPLIANT($c_n$) then
10:     EXPLORECOMPLIANT($T_s, c_n, CW$)
11:     EXPLORECOMPLIANT($T_s, c_n, CCW$)
12:     RETURN $c_n$
13: else
14:     RETURN $c_n$ {$c_r$ reached by PUSH}
```

B. Reverse local planner

In line 11 of Algorithm 1 a connection is tried from configuration c'_n in T_g to c_c in T_s. As explained in Section III, if a connection is made to the goal tree T_g, we need a different local planner because edges need to be directed *to* tree T_g as opposed to *away* from tree T_s in the previous section. A straightforward solution would seem to use the same local planner and just reverse the endpoints (starting at c_c and moving to c'_n). If however an obstacle blocks the path, no connection to T_g is found and we only extend T_s which undermines the strength of the bidirectional approach (e.g. escaping narrow passages) and makes the algorithm unsuited for an approach that uses multiple trees.

Algorithm 3 LOCALPLANNERREVERSE(T_g, c'_n, c_s)

```
1:  $c'_n \leftarrow$ CREATECONTACTTRANSIT($c'_n, c_s$)
2:  if $c'_n$ = NULL then
3:      RETURN NULL {failure}
4:  $c'_n \leftarrow$ REVERSEPUSH($c'_n, c_s$)
5:  if PUSHERCOMPLIANT($c'_n$) {did $P$ hit an obstacle during
    the reverse push?} then
6:      $c'_n \leftarrow$ REVERSEHOCKEYSTICK($c'_n$)
7:      if $c'_n$ = NULL then
8:          RETURN $c'_n$ {failure}
9:  if COMPLIANT($c'_n$) then
10:     REVERSEEXPLORECOMPLIANT($T_g, c'_n, CW$)
11:     REVERSEEXPLORECOMPLIANT($T_g, c'_n, CCW$)
12:     RETURN $c'_n$
13: else
14:     RETURN $c'_n$ {$c_s$ reached by REVERSEPUSH}
```

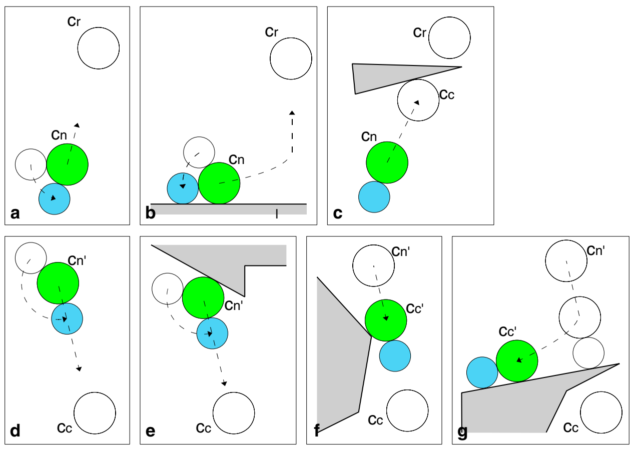

Fig. 8. The different situations the local planner has to deal with, (a..c) are the situations of the forward local planner, (d..g) are the situations of the reversed local planner. Note that in the reversed local planner the edges in the tree are directed opposite of the dotted arrows that show the direction of exploration.

Because of the above considerations, we need a true reverse version of the local planner. Because the edges are directed to T_g, the position of P is revolved 180 degrees (i.e. P precedes O during planning). Therefore it makes no difference whether c'_n is non-compliant (Fig. 8d) or compliant (Fig. 8e). We only need to verify whether P can reach the desired push position using a contact transit.

Next, the path (c'_n, c_c) is checked for collision. Three situations can occur. First, if c_c (which is part of T_s) is reached, a path from T_s to T_g is found and the algorithm ends. Second, on the path (c'_n, c_c) a collision of O with an obstacle can occur

(Fig. 8f) and a new compliant configuration c'_c is found. The third situation that can occur is that P collides with obstacle l' (Fig. 8g). A "reverse" hockey stick curve (or multiple curves as in Fig. 7) is used to see if a path from l' to c_n exists. If this succeeds, again a new compliant configuration c'_c is found.

If c'_c is compliant, the reverse exploration algorithm needs to be executed. Because the edges of T_g need to be directed to T_g we cannot use the same exploration algorithm but rather need a reversed version that is explained in Section V.

The complete algorithm is shown as Algorithm 3.

C. *Geometric primitives*

The local planner can be efficiently implemented using ray shooting and calculation of intersections (see [17]). To check a hockey stick curve for collision for both P and O requires numerical analysis. This can quite easily be implemented using techniques from e.g. [6].

In lines 5 and 10 of Algorithm 1 we need to find the configuration in the tree nearest to the random configuration. This nearest configuration is not necessarily a vertex of the tree, but can also be a position on an edge. Because it is not essential to find the nearest neighbor configuration exactly and because our edges do not solely consist of straight lines, we can use an approximate solution. Every edge in the tree is approximated by adding intermediate configurations along the edge that are used only for nearest neighbor searching [11].

V. Compliant exploration

If the local planner encounters an obstacle l at configuration c, compliant exploration is used to capture the topological structure of the set of compliant configurations that can be reached starting at c. The results of this exploration are compliant vertices that are added to the RRT (which strictly speaking becomes a graph). First we need to check if l has been explored before. If not, a vertex v at configuration c is added to the tree and we check which part of the compliant space can be reached from the current configuration. This procedure is called compliant exploration.

To capture the topology and to distinguish between explored and not yet explored compliant paths, intervals are used (see Definition II.6). An important property of an interval is that if it is reached, P is guaranteed to be able to push O to the end point c_e of that interval. Note that P is free to chose a push position within the push range as long as it is collision free and the bounding obstacles do not change. Exactly one interval is associated with every compliant vertex. This means that such a vertex does not represent a single configuration but rather a continuous subset of the compliant path of the obstacle (from its starting point to the end point of the interval).

Recall that an interval, among other properties, describes the bounding obstacles. As a consequence, an interval is unique to an exploration direction (in the other direction, the intervals are swapped). Note that at the same compliant position multiple intervals can be defined, it depends on the bounding obstacles of P to which interval the vertex belongs (Fig. 4).

Using the above observations, we can define which properties are needed to uniquely define a compliant vertex: *i)* the compliant interval to which it belongs and *ii)* the compliant position at which it starts. To check if the configuration of v belongs to an already explored part of the compliant space we only need to check whether the interval to which it belongs has been visited before. If this is not the case, v is used as a starting point to explore the compliant component in both the clockwise and counterclockwise directions. Since PR^+ and PR^- do not overlap, we will always need a contact transit for one of the exploration directions.

At the end of the interval, the exploration either ends or there is exactly one other interval we can reach (possibly after a contact transit). This other interval can be associated with the same obstacle, because the bounding obstacles have changed, but can also be associated to another obstacle that is part of the same compliant component. In either case, a new vertex v' is created, an edge (v, v') is added to the tree, and exploration is continued from v'.

There are a number of situations in which exploration ends. At the start of an interval, the contact transit to a position in PR^+/PR^- may fail. Also, if P is not able to push O any further along the compliant path because it gets stuck between two obstacles or is forced outside PR^+/PR^-, the exploration ends.

Exploration also ends if an interval is reached that has been encountered before. Because of the definition of an interval, from any position in the interval it is guaranteed that there is a push plan that pushes O to the end of the interval. As a consequence, if v and an existing vertex v_e are defined on the same interval, they can be merged by adapting the compliant start position of v_e to the compliant start position of v (there is no need to add v to the tree). No further exploration is necessary because v_e has already been explored. If O reaches the position where exploration started (e.g. it was pushed entirely around a compliant component) exploration also ends.

A. *Reverse exploration*

As stated before, if a connection is made from T_g, we need a reverse version of the compliant exploration algorithm because all edges need to be directed to the goal. Suppose the starting configuration c'_c of compliant exploration is connected to T_g, then the compliant paths that result from exploration need to be directed to c'_c. Thus for the compliant paths that are directed clockwise, the reverse exploration direction is counter clockwise and vice versa.

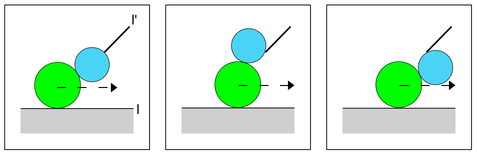

Fig. 9. Reverse exploration in the direction of the arrow. (a) At this point P has the choice between two intervals (either going clockwise or counterclockwise around l'). (b) If P chooses the counterclockwise direction, the path will quickly lead to a dead end. (c) If P chooses the path through the corridor between the two obstacles, reverse exploration can continue.

Reverse exploration is similar to forward exploration except when the end of an interval is reached. If at this configuration multiple intervals are defined, in contrast to forward exploration, more than one interval may be reachable. Luckily all but one of these paths will quickly lead to a dead end. This can be seen as follows (Fig. 9): if a new obstacle l' becomes a bounding obstacle, O either fits or does not fit between the first encountered point of l' and l. If it does not fit, then there will be a point where O hits both l and l' and a compliant path from l' to l may be possible at that point. If it does fit, no compliant path from l to l' at that point is possible. Depending on the result of this discriminant, the path of P can be chosen.

B. Geometric Primitives

Compliant exploration is a pure geometric process. Detecting to which interval a compliant configuration belongs or calculating where an interval ends and where a new one starts only involves finding intersections between line segments, disks and circular arcs. These are part of various Minkowski sums of the obstacles with disks of different radii. During a compliant motion, an obstacle can only enter PR^+/PR^- via either $\text{PR}_b^+/\text{PR}_b^-$ or $\text{PR}_e^+/\text{PR}_e^-$. This property can be used to find the compliant positions at which an interval ends and a new one starts. All intersections can be preprocessed and stored with their corresponding obstacles. Storing the line segments and circular arcs in a Kd-tree [5] allows for fast detection.

VI. Probabilistic completeness

The basic RRT algorithm (that only grows one tree from the start configuration) is known to be probabilistically complete. The advantage of the bidirectional version of the RRT (and thus also of reverse exploration) is that it helps escaping narrow passages. If the single-tree-RRT is used for pushing through open space, the position of P at a vertex is dictated by the direction O came from. Since every direction is possible (because of randomness) every pushable path through open space will eventually be found.

The above also means that every possible compliant configuration reachable from the open space will be found, because the first configuration of a compliant path is always preceded by a non-compliant path. Also since eventually every vertex will be generated, every hockey stick curve will be considered, and thus all possibilities to leave a compliant path will be tried.

VII. Experiments

Our algorithm has been implemented in Visual C++. Even without the use of Kd-trees, the preprocessing (calculating the intervals) is very fast. In a typical scene, for example the one of Fig. 1, the preprocessing took only 0.06s on a Pentium 2.4GHz. The query time in this scene was 0.01s on average (depending on the random choices of the RRT algorithm). In other scenes we observed similar running times.

VIII. Conclusions

In this paper we presented an algorithm that is capable of pushing a disk amidst obstacles using compliance. Using compliance extends the range of problems for which a successful push plan can be created. Also in environments or in subsets of environments in which the density of the obstacles is high, compliance helps in lowering the complexity of the solution. Using the RRT algorithm provides a natural balance between the number of compliant and non-compliant vertices in the trees. Since our algorithm supports both forward and reverse edges, it is also suited for a multi-tree approach [18].

If the environment is preprocessed, then given a compliant configuration, it is easy to check to which interval that configuration belongs and thus what part of the compliant space is reachable from it.

In this paper we did not allow the pusher to lose contact with the object. Allowing these non-contact transits is a subject of future research. If a contact transit is not possible (because of collision of the pusher), a path for the pusher could be planned using a preprocessed roadmap in which the current position of the object is incorporated.

Acknowledgements

Part of this research has been funded by the Dutch BSIK/BRICKS project.

References

[1] P. K. Agarwal, J.-C. Latombe, R. Motwani and P. Raghavan, Nonholonomic Path Planning for Pushing a Disk Among Obstacles, *Proc. IEEE Int. Conference on Robotics and Automation,* 1997

[2] N. M. Amato, O. B. Bayazit, L. K. Dale, C. Jones and D. Vallejo, OBPRM: An Obstacle-Based PRM for 3D Workspaces, *Proc. Int. Workshop on Algorithmic Foundations of Robotics (WAFR), pp. 155-168,* 1998

[3] H. Arai and O. Khatib, Experiments with Dynamic Skills, *Proc. of 1994 Japan-USA Symposium on Flexible Automation, 81-84,* 1994

[4] B. Aronov, M. De Berg, A. F. Van der Stappen, P. Ŝvestka, and J. Vleugels. Motion Planning for Multiple Robots, *Discrete and Computational Geometry, 22:505-525,* 1999

[5] J. L. Bentley, Multidimensional Binary Search Trees used for Associative Searching, *Commun. ACM, 18:509-517,* 1975

[6] R. L. Burden, J. Douglas Faires, Numerical Analysis 7th ed., *Wadsworth Group,* 2001

[7] A. J. Briggs, An Efficient Algorithm for One-Step Planar Compliant Motion Planning with Uncertainty, *Algorithmica, 8(3):195-208,* 1992

[8] B. R. Donald, The Complexity of Planar Compliant Motion Planning under Uncertainty, *Proc. ACM Symp. on Computational Geometry,* 1988

[9] M. A. Erdmann, On Motion Planning with Uncertainty, *Technical Report AITR-810 MIT Artificial Intelligence Laboratory,* 1984

[10] K. Goldberg, Orienting Polygonal Parts without Sensors, *Algorithmica, 10(2):201-225,* 1993

[11] S. M. LaValle, Planning Algorithms, *Cambridge University Press (or http://msl.cs.uiuc.edu/planning/),* To be published in 2006

[12] S. M. LaValle and J. J. Kuffner, Rapidly-exploring random trees: Progress and prospects, *B. R. Donald, K. M. Lynch, and D. Rus, editors, Algorithmic and Computational Robotics: New Directions, pages 293–308, A K Peters, Wellesley, MA,* 2001

[13] T. Lozano-Peréz, M. T. Mason and R. H. Taylor, Automatic Synthesis of Fine-Motion Strategies for Robots *IEEE Trans. on Computers (C-32),108-120,* 1983

[14] M. T. Mason, Mechanics of Robotic Manipulation, *MIT Press,* 2001

[15] M. T. Mason, Mechanics and Planning of Manipulator Pushing Operations, *Int. Journal of Robotics Research, 5(3):53-71,* 1986

[16] M. T. Mason, K. M. Lynch, Dynamic Manipulation, *Proc. IEEE/RJS Int. Conference Intelligent Robots and Systems, pages 152-159,* 1993

[17] D. Nieuwenhuisen, A. F. van der Stappen and M. H. Overmars, Path Planning for Pushing a Disk using Compliance, *Proc. IEEE Int. Conference on Intelligent Robots and Systems, pages 4061–4067,* 2005

[18] M. Strandberg, Augmenting RRT-planners with local trees, *Proc. IEEE Int. Conference on Robotics and Automation, pages 3258–3262,* 2004

[19] M. Sharir and S. Sifrony, Coordinated Motion Planning for Two Independent Robots, *Proc. of the fourth annual symposium on Computational Geometry, pages 319-328,* 1988