An Effective Framework for Path Planning amidst Movable Obstacles

Dennis Nieuwenhuisen, A. Frank van der Stappen, and Mark H. Overmars

Institute of Information and Computing Sciences, Utrecht University, The Netherlands, {dennis, frankst, markov}@cs.uu.nl

Abstract: This paper addresses the problem of navigating an autonomous moving entity in an environment with both stationary and movable obstacles. If a movable obstacle blocks the path of the entity attempting to reach its goal configuration, the entity is allowed to alter the placement of the obstacle by manipulation (e.g. pushing or pulling), to clear its path. This paper presents a probabilistically complete framework for solving path planning problems among movable obstacles. Heuristics are presented to provide efficient solutions for problems in environments encountered in practical situations.

1 Introduction

Motion planning [8] has been an active area of research for three decades. Over the past years the motivation has gradually extended from the traditional robotics context toward applications in computer-assisted training and advanced games. These applications feature planning problems of huge complexity, as they involve many (often human) entities with large numbers of degrees of freedom. The entities move in environments that are not necessarily fixed but can or even *must* (in the light of the objective of the training or game) be modified by the entities. As complete solutions (that guarantee a solution provided one exists) to motion planning are only feasible for problems involving a few degrees of freedom, research has led to approaches that provide a weaker form of completeness. A particularly successful approach that is suited for complex problems is the probabilistic roadmap method [6] in which a roadmap of the free space is built incrementally. After creation, the roadmap can be queried for a collision-free path for the moving entity. Over the past decade, many planners based on this principle have been proposed.

We explore the relatively unaddressed problem of planning the motions of a moving entity in an environment inhabited by both stationary and movable obstacles. Our motivation comes from an ultimate wish to automatically generate visually-convincing motions for computer-controlled entities in virtual environments and games. A rigorous way to plan motions among sta-

tionary and moving obstacles would be to consider the problem in the high-dimensional composite configuration space of the moving entity and the movable obstacles. Unfortunately, in all but the simplest instances the complexity is too high to efficiently find a solution.

However, with our motivation in mind, we believe that the above costly approach is not required. A human entity moving in a realistic (e.g. office) environment will plan his or her motions on the basis of knowledge about the layout of the stationary features of the environment. While executing the path, the entity may encounter movable obstacles such as a chair or a trolley with supplies standing in the way, or a door that is closed. If the entity encounters such movable obstacles it will try to move them out of the way by manipulating them or by getting around them, so that it will be able to continue its predetermined path. Only if the required manipulations get truly complicated or require a lot of effort, the entity may start to explore alternative paths toward the goal. Since it is our aim to provide convincing motions of entities, we want the planner to follow a similar strategy.

The difference between our approach and that in the composite configuration space resembles the difference between *decoupled* [5] and *centralized* [10] planning for multiple robots. As in decoupled planning we approach the problem in a lower-dimensional configuration space taking into account stationary features only. The resulting path is subsequently adjusted to resolve collisions with non-stationary features.

Rearrangement planning [1,2] is a problem closely related to motion planning among movable obstacles. Here, an entity also navigates in an environment among movable obstacles, but the goal is not defined in terms of a configuration for the entity, but rather in terms of configurations for the movable obstacles. Even though rearrangement planning has had considerable attention over the years, motion planning among movable obstacles has not.

Wilfong [13] has shown that motion planning among movable obstacles is NP-hard. Chen and Hwang [4] created a grid based planner that heuristically tries to minimize the cost to move obstacles out of the way. To reduce the cost, they only consider a very limited number of different states. With their planner they are able to solve some simple but realistic problems.

Another planner is the one developed by Stilman and Kuffner [12]. Their global approach uses the fact that the free space of the entity consists of multiple connected components. If start and goal are not in the same connected component then the entity uses manipulation to move obstacles to try to join connected components. To detect if a manipulation action has succeeded, a grid based approach is used. To manipulate an obstacle, contact points are sampled and a set of primitive actions is applied to those points. The candidate obstacles for manipulation are found by using an A^* search on a grid from the current position of the entity to the goal. Obstacles encountered during this search are the candidates. In case of failure, backtracking is used. The authors prove that their planner is resolution complete for a class of problems

they call LP_1, analogous to the LP_1 class in rearrangement planning [2]. The LP_1 class contains problems in which disjoint components of the free space can be merged by moving a single obstacle. Stated differently, only if an obstacle blocks the path of the entity directly, it will be manipulated. Problems that require the manipulation of an obstacle that blocks another obstacle will not be solved. An example of an LP_1 problem is shown as Figure 1a.

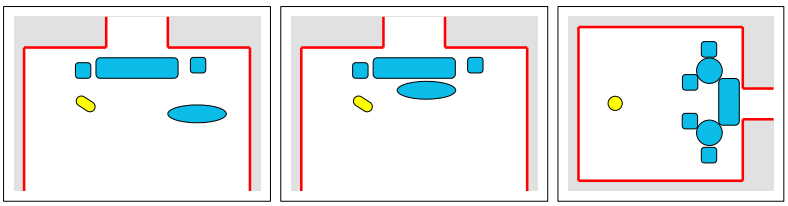

Fig. 1. Rooms consisting of couches, tables and chairs. The entity is represented as the light colored object. The goal for the entity is to leave the room. (a) Moving only the couch is enough to open the path to another room (LP_1). (b) First the table has to be moved before the couch can be moved (LP_2). (c) The couch is blocked by the round tables which on their turn are blocked by the chairs (LP_3).

Solving a motion planning problem among stationary and movable obstacles can be regarded as finding an alternating sequence of motions toward a movable obstacle and manipulation of these obstacles until the goal configuration is reachable without further manipulation. In this paper a novel framework is presented based on the expansion of a so-called action tree. This action tree represents the complete set of motions toward movable obstacles and manipulations of movable obstacles in every order. Finding a motion plan is equivalent to finding a path in the action tree. If the problem gets more complicated (i.e. more movable obstacles are involved), the construction of the complete action tree may become infeasible. Therefore, we present heuristics inspired by problems encountered in practical applications that guide the construction process of the action tree in favor of promising tree nodes. The class of problems that can be solved using our planner is LP [2], i.e., the set of problems that can be solved by a sequence of manipulations (Figures 1b+c). In contrast to the LP_1 class, the LP class contains problems for which movable obstacles have to be manipulated that do not directly block the path of the entity. For example, the manipulation of a movable obstacle can be blocked by another movable obstacle for which the manipulation is blocked by yet another movable obstacle etc. During the manipulation of a movable obstacle, the other movable obstacles are assumed to be stationary.

This paper is arranged as follows, first in Section 2, we will describe the problem in more detail and state some properties for the solution. Next, in Sections 3 and 4 the action tree and its properties are presented. The heuristics

to guide the search process are described in Section 5. Finally, results of our experiments in are presented in Section 6.

2 Problem statement and preliminaries

Let E be an entity defined in a workspace that, besides stationary obstacles, contains k movable obstacles (or movables for short) $M = \{M_1, M_2, ...M_k\}$. All movables are assumed to be closed sets. A movable M_i cannot move by itself, but can only be moved by E if M_i is first grasped by E. The distance between two objects O_i and O_j is denoted by $d(O_i, O_j)$, which is the Euclidean distance between the two points on the boundary of O_i and O_j that are closest together. If the Euclidean distance between points p and q is denoted by $e(p, q)$ and $\text{INT}(O)$ denotes the interior of object O, then:

$$d(O_i, O_j) = \begin{cases} \min\limits_{p \in O_i, q \in O_j} e(p,q) & \text{IF } \text{INT}(O_i) \cap \text{INT}(O_j) = \emptyset \\ \infty & \text{IF } \text{INT}(O_i) \cap \text{INT}(O_j) \neq \emptyset \end{cases}$$

M_i is only said to be grasped by E if $d(E, M_i) = 0$. If M_i is grasped by E, the combination of E and M_i is denoted by M_i^E. A physical model is used to define the set of possible motions of M_i^E, depending on the forces that E is able to apply to M_i. This model can also describe the potential positions on the boundary of M_i where grasps are allowed. Since the physical model is highly dependent on the specific capabilities of E and the properties of the environment, we use a simplified model in which E is capable to push/pull a movable in any direction and grasp it whenever $d(E, M_i) = 0$. Other models can be used without affecting the algorithm. We do not allow E to manipulate two movables at the same time.

Our goal is to create a motion plan for E from a given start to a given goal configuration. The behavior of E should be convincing compared to the behavior of its real (e.g. human) counterpart. For example, if a human has the choice between moving multiple obstacles that block a door and taking a small detour, it will most likely do the latter. Also most problems will be solved locally, i.e. a blocking movable will not have to be pushed a long distance before E will be able to move around it.

In this paper we introduce the concept of an action tree. The action tree is a general framework for solving motion planning problems among movable obstacles. In contrast to a roadmap graph, which represents the free configuration space, the action tree represents the different actions E can perform given the configurations of the movables. Using this framework, a planner is presented that uses heuristics to guide the search process through the action tree in order to efficiently find a solution.

3 Action tree

In the basic motion planning problem all obstacles are stationary. Here, the movables can also change position during the execution of the planning al-

gorithm. Therefore we introduce the notion of a *worldstate* that encodes the placement of all non-stationary obstacles, i.e. entity E and movables M.

Definition 1 (Worldstate). *A* worldstate $W = (w_e, w_1, ..., w_i, ..., w_k)$ *describes the configuration of E and the configurations of all movables in M.*

A worldstate is essentially a point in the composite configuration space of E and M. As stated before however, we will not solve the planning problem in this composite configuration space.

We will define two basic actions that are used to transform one worldstate into another. The first action is GRASP(M_i). A successful call to GRASP(M_i) transforms worldstate $W = (w_e, w_1, ..., w_i, ..., w_k)$ into worldstate $W' = (w'_e, w_1, ..., w_i, ..., w_k)$ satisfying $d(E[w'_e], M_i[w_i]) = 0$, else it reports failure. GRASP(M_i), moves E from its current configuration w_e to a randomly selected configuration w'_e on the boundary of M_i. If M_i cannot be grasped (because E is not able to reach the boundary of M_i) the action reports failure. Note that if $d(E[w_e], M_i[w_i]) = 0$, the GRASP(M_i) action has the effect of re-grasping M_i at another location. A re-grasp can be useful if the current grasp does not suffice, for example if the room for M_i^E to maneuver is limited.

The second action is MANIP(M_i^E), which tries to manipulate the currently grasped movable. A successful call to MANIP(M_i^E) transforms worldstate $W = (w_e, w_1, ..., w_i, ..., w_k)$ with $d(E[w_e], M_i[w_i]) = 0$ into worldstate $W' = (w'_e, w_1, ..., w'_i, ..., w_k)$ with $d(E[w'_e], M_i[w'_i]) = 0$. MANIP($M_i^E$) results in a joint motion of M_i and E satisfying the constraints imposed by the physical model from their current configuration to a randomly selected configuration. If MANIP(M_i^E) does not succeed, for example because the physical model forbids the manipulation or a collision occurs, MANIP(M_i^E) reports failure.

We now define the *action tree* T_A that represents all valid actions in all possible orders.

Definition 2 (Action Tree). *The* action tree *is a description of the space of all valid actions. Every node of the tree corresponds to a worldstate. The edges between the nodes represent the action* (GRASP() or MANIP()) *that results in the transformation from one worldstate to another worldstate.*

At every node n of T_A, a worldstate $W(n)$ is associated. T_A consists of two types of nodes: *manipulation nodes* are the result of a call to MANIP(), *grasp nodes* are the result of a call to GRASP(). A manipulation node will never have a child node that is also a manipulation node. This can be seen as follows. Suppose manipulation node n has a child node m that is also a manipulation node. Then node m could also have been a direct child of the parent of n (e.g. a sibling of n). The same holds for grasp nodes, a grasp node will never have another grasp node as a child because the second movable should have been grasped at once without first grasping the first one. Therefore a grasp node will always have a manipulation node as a parent and vice versa. We do not allow E to grasp two movables at the same time.

After initializing the algorithm by associating the initial worldstate to the root node n_s of T_A, the algorithm constructs T_A by *expanding* nodes. The expansion of a manipulation node adds a child node in which a movable is grasped at a random configuration; this may also be a re-grasp of the currently grasped movable. The expansion of a grasp node adds a child node that manipulates the currently grasped movable.

After the addition of a manipulation node n to T_A, E may be able to reach its destination. If a grasp node is added, this is not possible since no movables have changed their configuration. After manipulating a movable though, a new path may have emerged that brings E to its goal. Therefore after the addition of a manipulation node to T_A, the algorithm checks whether E can reach its goal configuration. If this succeeds, the algorithm terminates, if not, the construction of T_A continues. An example of the construction of an action tree is shown in Figure 2.

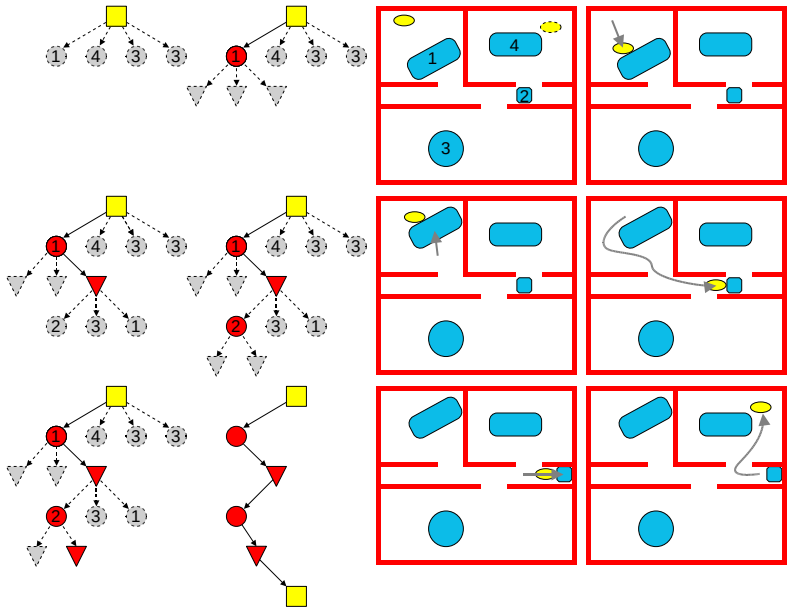

Fig. 2. Example of the construction of T_A. Left the the action tree T_A is shown, at the right the corresponding workspaces. The circle nodes in T_A are the grasp nodes, the triangles are the manipulation nodes. The square nodes are the start and goal configurations. The dashed nodes are omitted parts of T_A that did not contribute to the final solution or were unfeasible. In the first workspace, the start position of E is shown in the top left, the goal in the top right.

A motion plan is a finite alternating sequence of grasp and manipulation actions. In the action tree such a plan is represented by a path from the root to another node. If the physical model is such that during a manipulation M_i^E behaves as a rigid body and the physical model imposes no nonholonomic con-

straints on the motion, then, eventually every possible grasp and manipulation action will be represented in T_A. Also if nodes in T_A are selected randomly for expansion, every possible order of such actions will be represented by a path in T_A from the root node to another node. Therefore the framework of the action tree is probabilistically complete.

4 Realization of the action tree

In this section we will describe how to implement the necessary building blocks for the action tree. In scenes encountered in practical settings, we have observed that after a few expansions of a node, further expansion rarely led to more progress. For that reason we only expand nodes once (i.e. only leaves in T_A are expanded) but add multiple child nodes at once. If a manipulation node is selected for expansion, k children are added all grasping a different movable (including the current one). The expansion of grasp node adds a number of child nodes that manipulate the currently grasped movable.

To grasp a movable or to check whether the goal configuration can be reached, a graph G is used. G represents the free space for E with respect to the stationary obstacles. It should preferably contain cycles to provide alternative routes. Any path planning technique that results in a graph for E that represents feasible paths can be used to create G. A well known example of such a technique is the probabilistic roadmap method [6]. Without loss of generality, we will assume that the start and goal configurations for E are configurations in G.

4.1 Checking if the goal can be reached

After the addition of a manipulation node n, E may be able to reach its destination. Since G is collision free for the stationary obstacles only, it needs to be updated according to $W(n)$. Edges in G that collide with one of the movables have to be invalidated such that queries are guaranteed to be collision free.

A manipulation transforms worldstate $W(p(n)) = (w_e, w_1, ..., w_i, ..., w_k)$ to worldstate $W'(n) = (w'_e, w_1, ..., w'_i, ..., w_k)$, where $p(n)$ is the parent of node n. To be able to quickly update G for a given worldstate, at every manipulation node, a list of invalidated edges of G is stored. This list is equivalent to the list of $p(n)$ except for edges that intersect with either $M_i[w_i]$ or $M_i[w'_i]$. Only these edges have to be collision checked against M_i. To find these efficiently, the endpoints of the edges are stored in a Kd-tree [3]. A Kd-tree allows for quickly identifying the edges that are close to w_i so that they can be checked for collision. When the root node is added to T_A, G needs to be checked once against all movables in order to create the initial list of invalidated edges. Grasp nodes simply copy the list from their parent. After updating G, w'_e is connected to G using standard procedures from motion planning. Finally, a query is performed to see if E can reach its destination.

4.2 Grasping a movable obstacle

The GRASP($M_i[w_i]$) action creates a path for E from its current configuration w_e to w'_e where $d(E[w'_e], M_i[w_i]) = 0$. Here, w'_e is a randomly chosen grasp on the boundary of M_i. Depending on the application, the selection of grasp configurations can be customized. As described in the previous section, G is updated w.r.t. the current worldstate. Next, w_e and w'_e are attempted to be connected to G. If this succeeds, a query can be executed between w_e and w'_e. If the query is successful, a new node is added to T_A.

4.3 Manipulating a movable obstacle

After E has grasped M_i, resulting in a grasp node, it will try to manipulate M_i by executing the MANIP(M_i^E) action. To move M_i^E we will use an approach based on the Rapidly-exploring Random Trees (RRT) algorithm [9]. An RRT is aimed at growing a tree from a given start configuration in an attempt to cover the free space. Here, the RRT operates in the configuration space of M_i^E where all movables M_j with $i \neq j$ are considered stationary. An RRT quickly generates many different paths away from the start configuration. This property is very useful in our situation because our target is to move M_i^E away from its current configuration to many different goal configurations.

The vertices[1] in the RRT represent configurations for M_i^E. The edges represent paths between them. The RRT algorithm works as follows. First the start configuration is added to the RRT as a vertex. Next, a random (not necessarily collision free) configuration $c_r = (w_e, w_i)$ with $d(E[w_e], M_i[w_i]) = 0$ for M_i^E is generated. The nearest configuration $c_n = (w'_e, w'_i)$ in the RRT to c_r is found (not necessarily a vertex of the RRT) and a path is tested for collision moving from c_n to c_r. If c_r is reached, it is added as a vertex to the RRT together with the edge (c_n, c_r). If a collision occurs before c_r is reached, the last collision free configuration $c_s = (w''_e, w''_i)$ is added to the RRT together with the edge (c_n, c_s). This process is repeated until some stop criterion is met.

The RRT algorithm needs to check whether a path exists between c_n and c_r. To verify the existence of such a path a *local planner* is used in many sampling based motion planning techniques. Given two configurations the local planner checks whether the path between them is feasible. Because of the many collision checks involved, a call to a local planner may be relatively expensive. Therefore, usually the local planner only checks whether the straight line connection between two configurations is feasible. Rotational parameters are often interpolated. In our algorithm, the generated path also needs to comply with the physical model for the specific type of manipulation (e.g. pushing/pulling). The local planner is allowed to use any physical model as

[1] Note the difference between the action tree that contains nodes, and the RRT that contains vertices.

long as it is capable of deciding whether a manipulation path exists between two configurations or, in case of a collision, what the closest reachable configuration is to c_r.

Since the RRT algorithm works incrementally by nature, it is easy to implement our MANIP() action using an RRT. Every grasp node n contains an RRT. The child nodes of n (which are all manipulation nodes) represent the vertices of that RRT.

We have now described all the building blocks necessary for expanding a node. Algorithm 1 shows how a node is expanded. Calling this function with n_s (that contains the initial worldstate), initiates the creation of T_A. Nodes are expanded in a breadth first manner to provide an optimal solution. In the next section, we will describe heuristics to tailor this concept to problem settings encountered in practical settings.

Algorithm 1 EXPANDNODE (n_s, G, W_g)

```
1: if n_s.type = MANIPULATIONNODE then
2:   for all M_i ∈ M do
3:     W' ← GRASP (M_i, W(n_s)) {grasp M_i at a random position}
4:     if W' ≠ NULL then
5:       n_s.AddChild(W') {add a grasp node}
6: else {n_s is a grasp node}
7:   n_s.InitRRT() {n_s is the container of the RRT}
8:   for i = 0 to MAXRRTVERTICES do
9:     W' ← EXTENDRRT (W(n_s))
10:    if W' ≠ NULL then
11:      n_s.AddChild(W') {add a manipulation node}
12:      UPDATEENTITYGRAPH(G, W') {update G for W'}
13:      if PATHEXISTS(G, W', W_g) {is there a path to W_g?} then
14:        return PATHFOUND {the goal can be reached}
15: for i = 1 to n_s.nrChildren do
16:   EXPANDNODE (n_s.Child[i], G, W_g)
```

5 Planner

Although breadth-first expansion of T_A guarantees an exhaustive exploration of all possible sequences of manipulations, this strategy becomes computationally expensive or even infeasible when the number of movables is large. In problems encountered in practical settings however, only a small subset of the movables are involved in the final motion plan of E. In that respect, many nodes of T_A will most likely not contribute to the final solution. For example, if a motion plan needs to be created that moves E from room A to B, then often the movables present in room C will not be part of the final motion plan. On the other hand, a movable that is not directly impeding the path of E, may very well be blocking the manipulation of another movable and because of that be part of a feasible motion plan. Because of the above considerations, expanding nodes in a breadth first search manner is not the

most efficient way to find a motion plan. In this section we will describe how to focus the expansion process toward promising nodes, such that a solution can be found rapidly without affecting the probability of finding a solution.

5.1 Choosing a path through the action tree

Instead of expanding T_A in a breadth first manner, we will use heuristics to guide the expansion process. To be able to do this, every node n is assigned a probability $q(n)$. The following holds:

$$\sum_{m \in \text{children}(n)} q(m) = 1 \tag{1}$$

Nodes are now selected for expansion by creating a path from the root node to a not yet expanded node (a leaf). Starting at the root (having probability 1), a child node is selected randomly based on its probability. This process is repeated until a leaf is reached. Then that leaf is selected for expansion.

After expanding a node n, all its children are initially assigned equal probabilities. Later on in the process we will increase probabilities for a node depending on the progress that is made. For example, if n is successful in getting closer to the goal, $q(n)$ is increased. Increasing $q(n)$ should not violate Equation 1. In addition we must assure that $q(n)$ never reaches 0 as this would exclude its selection. If $q(n)$ is already very high (e.g. $q(n) = 0.95$), there is little reason to increase it more. Using the above observations, we use the following procedure to adapt the probabilities: a fraction $f \in [0, 1]$ is used to increase $q(n)$ of a node n. We denote the updated value of the probability of n by $q'(n)$. The siblings of node n are denoted by the set $S(n)$.

$$q'(n) = (1 - q(n))f + q(n) \tag{2}$$

$$\underset{m \in S(n)}{\forall} : q'(m) = q(m)(1 - f) \tag{3}$$

Similarly if, after the expansion of a node, no or little progress is made, the probabilities of selecting that node should be decreased. The following procedure is used to decrease the probabilities of node n if n has at least one sibling:

$$q'(n) = q(n) - f \cdot q(n) \tag{4}$$

$$\underset{m \in S(n)}{\forall} : q'(m) = \frac{1 - q(m)}{|S(n)| - 1 + q(n)} f \cdot q(n) + q(m) \tag{5}$$

Equations 2-5 lead to the following lemma, for which the proof is omitted:

Lemma 1. *The updates given by Equations 2 and 3 and by 4 and 5 maintain Equation 1.*

The probability that n will be selected for expansion is: $q(n) \cdot \prod_{m \in \text{ancestors}(n)} q(m)$. If $q(n)$ is increased (at the expense of its siblings), the probability of n being selected will increase only by a small amount (especially when n is not close to the root). To solve this issue, the increased probability of n is *propagated* along the path from n to the root, i.e. the probabilities of all ancestors of n are increased as well. We must make sure however that after a few updates, the probabilities of nodes higher in the tree do not become too high. If we consider the path from n to the root, the further away a node n' is from n, the less similarity between $W(n)$ and $W(n')$. Therefore f is lowered during the propagation. The procedures to increase and decrease the probabilities of nodes are shown as Algorithms 2a and 2b.

Algorithm 2 (a) Increasing and (b) decreasing probabilities

IncreaseProbability (n, f)	DecreaseProbability (n, f)
1: $q'(n) = (1 - q(n)) \cdot f + q(n)$	1: **if** $\|S(n)\| > 1$ **then**
2: **for all** $m \in S(n)$ **do**	2: $q'(n) = q(n) - f \cdot q(n)$
3: $q'(m) = q(m) * (1 - f)$	3: **for all** $m \in S(n)$ **do**
4: $f = factor * f$ {lower f by a factor}	4: $q'(m) = \frac{1-q(m)}{\|S(n)\|-1+q(n)} + q(m)$
5: IncreaseProbability $(p(n), f)$	5: $f = factor * f$ {lower f by a factor}
	6: DecreaseProbability $(p(n), f)$

An example of increasing the probability of a node is shown in Figure 3. Since only the nodes (and their children) on the path from n to the root are updated, the cost of the propagation is $O(r)$, where r is the rank of n.

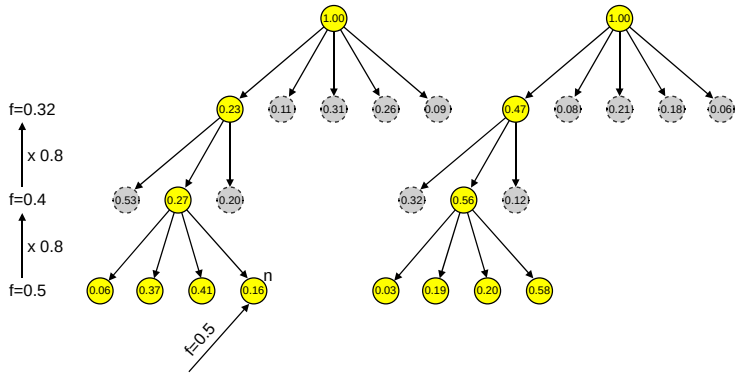

Fig. 3. Increasing the probability of a node. The values in the nodes are the probabilities. (a) The probability of node n is increased by a fraction of 0.5. If f is propagated to the parent node, it is multiplied by a factor of 0.8 (shown left). (b) The resulting probabilities after propagation using Algorithm 2a.

The value of the fraction f determines the global behavior of the algorithm. If f is small, the differences between the probabilities of the nodes will not be large, resulting in a breadth first type of expansion. If f is high however,

a small number of nodes will receive a high probability, resulting in a more depth first type of expansion.

5.2 Adapting probabilities

As stated before, often only a small subset of M will be blocking the path of E. Therefore we will increase the probability of manipulating movables that actually block the path of E. A movable M_i can block the path of E either *directly* or *indirectly*. Directly blocking means that the path of E actually collides with M_i, indirectly blocking means that some M_j, $i \neq j$ blocks the manipulation of M_i.

Directly blocking movable obstacles
After the expansion of a manipulation node n, it is checked whether E can reach its destination (Section 4). If no path to the goal is found, it is necessary to manipulate movables. Selecting a good movable candidate for manipulation involves taking into account our target of creating convincing paths for E. One of the properties of such a path is that a small detour is favorable over manipulating many movables. Therefore a second version of G is created in which no edges are invalidated but rather get a penalty when colliding with one of the movables. This graph is denoted by G^p. The cost of traversing an edge in G^p is a function of its length in C-space and the penalty. Using G, G^p can be constructed by assigning a penalty to edges instead of invalidating them. No additional collision checks are necessary to construct G^p.

After reconnecting E to the graph, a shortest path query on G^p, using for example A^* or D^* Lite ([11], [7]) yields a path that prefers to avoid movables. By using the result of the query, the *first colliding movable* M_i on the path to the goal in G^p can be easily determined. M_i is called a *directly blocking movable*. The probability of the child node of n grasping M_i is increased.

The above procedure is repeated a few times to make sure that the expansion process does not become too focused toward one path. For this, the edges in G^p that collide with M_i are invalidated and a new query is initialized. If successful, again the first colliding movable is determined and the corresponding node's probability is increased.

Indirectly blocking movable obstacles
Movables that block the path of E directly, can be determined by a shortest path query. However, a movable can also block the manipulation action of another movable, thus blocking the path of E indirectly (see Figure 4a for an example). Using the properties of the RRT algorithm, these can be determined easily. Recall that an RRT is extended by trying to create a path for M_i^E from the RRT to a randomly chosen configuration. If this random configuration is not reached, M_i^E collides with either a stationary obstacle or another movable M_j (Figure 4b). In the latter case, M_j blocks the manipulation path of M_i^E and M_j is identified as an *indirectly blocking movable*.

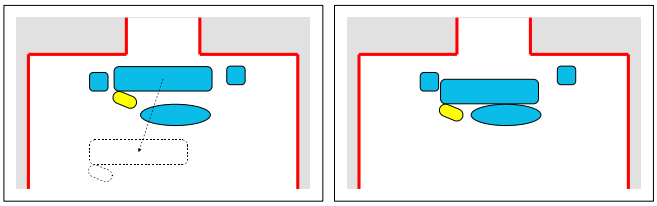

Fig. 4. E tries to pull the couch to the dotted target position. (a) The start of the manipulation action. (b) The coffee table blocks the path. This configurations is added to the RRT and the probability of the node in which the coffee table is grasped is increased.

During the expansion of a grasp node n in which M_i is grasped (creating child nodes that are vertices of the RRT), if a blocking movable M_j is encountered, the probability of the sibling of n that grasps M_j is increased using Algorithm 2a. The more often M_j acts as indirectly blocking movable, the more likely it is that M_j is selected for manipulation.

Decreasing probabilities
If a node n in T_A has many successful descendants, $q(n)$ will increase because of the propagation algorithm. However, if its descendants cease to make progress, its probability should be lowered. For this, we need a method to measure progress. A cheap measure of progress is the graph distance of E to the destination in G. The current position of E is connected to G and using a shortest path query to the destination provides an estimate of the current distance to the goal. The distance estimate is saved in the node such that the progress between a node and its parent can easily be determined. If no or little progress is made, the probability of the node is decreased by a small fraction using Algorithm 2b.

5.3 Lazy expansion

Expanding a node (Algorithm 1) can be a costly operation. Luckily many operations can be postponed until the moment a node is actually selected for expansion. Manipulation nodes contain child nodes that grasp a movable. Checking the feasibility of the grasp (i.e. can the grasp configuration be reached by E from its current configuration) is not necessary until the moment the child node is selected for expansion. So if a manipulation node is chosen for expansion, it adds child nodes that represent grasps of movables without verifying that such grasps actually exists.

If a child node is added to a grasp node, the entity graphs (G and G^p) need to be updated. This update involves collision checks and is thus relatively expensive. Only when the child node is selected for expansion this update is necessary. Therefore the calculation of the update of G and G^p can be postponed until the node is actually chosen for expansion.

If lazy expansion is used then it is uncertain if n is feasible at the moment it is added to T_A. Only if n is selected by the random process for expansion,

its feasibility is checked. If it is not feasible, n is declared a *dead end* and $q(n)$ is set to 0. The probabilities of $S(n)$ are updated such that Equation 1 is maintained. Now another random leaf node is selected for expansion.

6 Experiments

We implemented our algorithm in C++ and conducted experiments with three scenes on a Pentium 2.40GHz with 1GB of memory. In all scenes the heuristics as described in Section 5 were used. The different values for f were determined experimentally and the same values turned out to be useful in all experiments. A node grasping a blocking movable received an increase of its probability using $f = 0.8$. The probabilities of nodes containing an indirectly blocking movable were increased using $f = 0.05$ for every collision. If a new node did not result in progress toward the goal, the probability of the corresponding node was decreased using $f = 0.2$. The number of child nodes added to a grasp node was at most 5, the edge penalty used in G^p was 100 times its length.

For the first experiment the LP_3 scene of Figure 1c was used. Next, for the second experiment, to verify that our method also scales to multiple indirectly blocking movables, we took three versions of the scene of Figure 1c and connected them together, effectively creating a problem involving a series of three LP_3 problems. The third scene is shown in Figure 5. The pitfall in this scene is that the shortest path (shown as the dotted arrow) is blocked by an immovable obstacle close to the goal. Thus, to reach a solution, the algorithm probably needs backtracking (depending on the random choices). The fourth experiment (Figure 6) has two indirectly blocking movables. For all scenes, we conducted 100 experiments and averaged the results, shown in Table 1.

Table 1. Results of the experiments. The results were obtained by averaging 100 runs. The average tree size is the average total number of nodes in the tree.

Scene	Avg. running time	Avg. tree size	Avg. rank of solution
1	1.06s	81	14.4
2	8.1s	167	43.8
3	29.0s	506	13.5
4	22.0s	400	23.3

7 Discussion

In this paper we have presented a probabilistically complete framework based on an action tree to solve motion planning queries among movable obstacles. The nodes of the tree represent worldstates, the edges the transitions between the worldstates. During those transitions, the entity either manipulates one of the movables or (re)grasps a movable. A path in the action tree represents a motion plan. Constructing the complete action tree (in a breadth first search manner) can be infeasible because of the huge number of nodes.

In environments encountered in practical settings often only a small subset of the motion plans in the action tree are useful. Therefore we presented

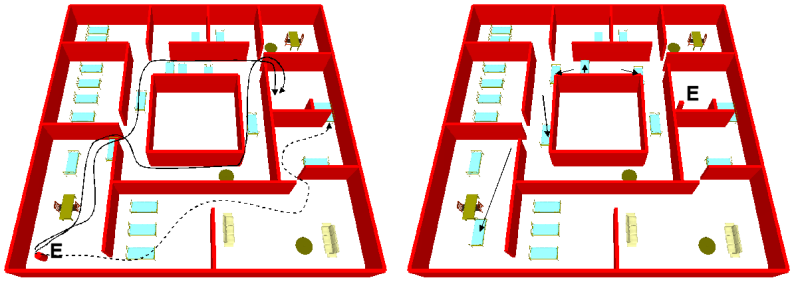

Fig. 5. The third scene. (a) The entity E is represented by a cylinder. The two solid arrows show feasible paths, the dotted path results in a dead end (behind the first movable is another one that blocks the manipulation of the first). (b) The situation after the entity has reached its destination.

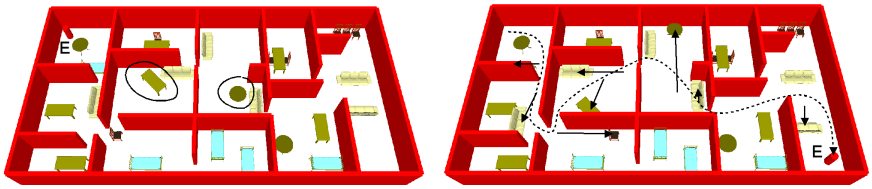

Fig. 6. The fourth scene. The entity needs to move from the top left to the right bottom. (a) The two encircled movables are indirectly blocking movables. (b) The situation after the entity has reached its destination.

heuristics that focus the node expansion process toward these motion plans. This process is solely guided by adapting the probabilities of selecting nodes. By continuously adapting these probabilities, using information gathered during the process, an efficient algorithm is obtained that is capable of solving realistic problems in reasonable running times.

An issue we did not address in this paper is smoothing. To smooth the path of the entity during a grasp action, standard smoothing techniques can be used. Also, since the final path is the result of a probabilistic process, some nodes from the action tree in the solution may be redundant. A smoothing procedure may be able to by-pass them.

Even though the heuristics are often successful in guiding the probabilistic process in realistic problems, in certain situations the process may become slow. This happens if many movables are close together and need to be manipulated in a certain order. If n movables are involved, there are $n!$ sequences of manipulating the movables. Since none of these result in overall progress, many nodes get comparable probabilities resulting in breadth first search behavior. Experiments have also shown that sometimes computation time is spent on parts of the action tree that are quite similar (e.g. they only slightly differ in the configuration of a movable).

A solution could be to use information gathered in the process not only locally in the action tree but rather in all nodes that resemble the current node in some aspects. For example, if a movable is impeding the entity in a node, then its probability of manipulation should not only be increased in that node but in all nodes that contain a similar subproblem. If the subproblem is solved in one node, then a shortcut in the action tree could be added to all other nodes. This type of extensions are the subject of current research.

Acknowledgments

Part of this research has been funded by the Dutch BSIK/BRICKS project. The authors would like to thank Jur van den Berg for fruitful discussions and suggestions.

References

1. R. Alami, J.P. Laumond and T. Siméon, Two Manipulation Planning Algorithms *Workshop on Algorithmic Foundations of Robotics, 109–125*, 1994
2. O. Ben-Shahar and E. Rivlin, Practical Pushing Planning for Rearrangement Tasks, *IEEE Trans. on Robotics and Automation, 14(4):549–565*, 1998
3. J. L. Bentley, Multidimensional Binary Search Trees used for Associative Searching, *Commun. ACM, 18:509–517*, 1975
4. P. C. Chen and Y. K. Hwang, Practical Path Planning among Movable Obstacles, *Proc. IEEE International Conference on Robotics and Automation, pp. 444–449*, 1991
5. M. Erdmann and T. Lozano-Pérez, On multiple moving objects, *Algorithmica, 2:477–521*, 1987
6. L. Kavraki, P. Švestka, J.-C. Latombe, M. H. Overmars, Probabilistic Roadmaps for Path Planning in High-Dimensional Configuration Spaces, *IEEE Trans. on Robotics and Automation 12, pp. 566–580*, 1996
7. S. Koenig and Maxim Likhachev, Improved Fast Replanning for Robot Navigation in Unknown Terrain, *Proc. of the 2002 IEEE International Conference on Robotics and Automation, pp. 968–975*, 2002
8. S. M. LaValle, Planning Algorithms, *Cambridge University Press* (or http://msl.cs.uiuc.edu/planning/), 2006
9. 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
10. J. T. Schwartz and M. Sharir, On the "piano movers" problem III: coordinating the motion of several independent bodies: the special case of circular bodies moving amidst polygonal barriers, *Int. Journal of Robotics Research, 2(3):46–75*, 1983
11. A. Stenz, The Focused D^* Algorithm for Real-Time Replanning, *Proc. of the International Joint Conference on Artificial Intelligence, pp. 1652–1659*, 1995
12. M. Stilman and J. Kuffner, Navigation Among Movable Obstacles: Real-time Reasoning in Complex Environments, *Int. Journal of Humanoid Robotics, 2(4):479–504*, 2005
13. G. Wilfong, Motion Planning in the Presence of Movable Obstacles, *Proc. of the 4th Annual Symposium on Computational Geometry, pp. 279–288*, 1988