Proceedings of the 2007 IEEE/RSJ International
Conference on Intelligent Robots and Systems
San Diego, CA, USA, Oct 29 - Nov 2, 2007

Efficient Path Planning in Changing Environments

Dennis Nieuwenhuisen Jur van den Berg Mark Overmars

Abstract— **This paper addresses the problem of path planning in environments in which some of the obstacles can change their positions. It uses the popular PRM method for navigating a robot through an environment. One of the key features of PRM is that it moves the major part of the calculations involved in the path planning process to the preprocessing phase. After that, paths can be extracted very quickly (in a query phase) usually without any noticeable delay. While very successful in many applications, doing most of the work in a preprocessing phase restricts the environment to be static i.e. obstacles are not allowed to change their configurations after the preprocessing phase. In this paper we describe and evaluate an algorithm based on PRM that does allow obstacles to change their configuration after preprocessing while still allowing for a quick query phase.**

I. INTRODUCTION

The Probabilistic Roadmap Method (PRM) [9] has become one of the leading path planning techniques in the field of robotics, both in virtual and real-world contexts. Recently its applications have extended to the domain of computer-assisted training, advanced gaming [12], [8] and biology (see e.g. [13]). Its main features are its simplicity, allowing for almost instantaneous queries, extensibility to higher dimensions and the broad range of problem types to which it is applicable. The PRM method works by sampling collision-free configurations and by connecting these by collision-free local paths (created by a *local planner*). A graph (the roadmap) is thus formed that aims at representing the connectedness of the free space. If the roadmap adequately represents this connectedness, a path between two collision-free configurations can be computed efficiently. An important property of the PRM planner is that the major part of the computations are done in a preprocessing phase. After this preprocessing phase, paths can be extracted quickly in a query phase allowing for interactive performance.

The PRM methods assumes the environment to be static. That is, between the creation of the roadmap and the query phase, the environment is not allowed to change since this could violate the premise that the roadmap represents the free space. However, realistic environments are often not static, but contain obstacles that change their positions over time. A straightforward solution to this problem is to update the roadmap after an obstacle has changed its position. Unfortunately such updates are usually computationally expensive because they involve collision checks. This detracts from the important property of the fast query phase. In this paper we propose an algorithm that makes the roadmap robust against changes in positions of some of the obstacles while still providing for a fast query phase. We call such environments *changing environments*. Besides stationary obstacles, changing environments contain moving obstacles that can have a different position for each query.

Path planning in changing environments is a relatively unaddressed problem. In [11] a voxel grid is used that covers the workspace and stores for each roadmap edge which voxels are intersected by the robot during the path associated with that edge. When an obstacle changes position, the voxels are used to identify the edges that are invalidated by the obstacle. This is done by checking the voxels covered by the obstacle for overlap with the swept voxels of each of the edges. Using only the edges of the roadmap that are not invalidated, a path can quickly be found.

A more ad-hoc approach is used in [7]. This planner first constructs a cycle-free roadmap for the static part of the environment. If, in a query, an edge crucial for finding a path intersects with a moving obstacle, a variant of the Rapidly-exploring Random Trees (RRT) approach [10] is used to reconnect the vertices of this edge. If this procedure fails, an attempt is made to reconnect the two disconnected components of the roadmap by additional global sampling.

The above methods aim at solving the problem in the query phase rather than in the preprocessing phase. We present an algorithm that creates a *robust roadmap* in the preprocessing phase by using the observation that the motion of the moving obstacles is often not unconstrained, but is restricted to some confined area. Examples of such obstacles are a door that can be open or closed or a chair whose position is bounded to a room. In a previous paper [1] we presented an algorithm that exploits this property by assuming that a moving obstacle has a predefined set of potential placements. While a straightforward implementation of this algorithm leads to roadmaps that are robust against placement changes of these moving obstacles, the algorithm suffers from some drawbacks.

In this paper we will first briefly describe the algorithm and identify its problems. We will conclude that the main problem is that the performance of the algorithm decreases quickly if the number of moving obstacles increases. Next we will pinpoint the cause of this problem and then state a solution for it based on Boolean logic. We will also present some heuristics to further improve the performance

Part of this research has been supported by the Dutch BSIK/BRICKS project and by the GATE project, funded by the Netherlands Organization for Scientific Research (NWO) and the Netherlands ICT research and Innovation Authority (ICT Regie).

Dennis Nieuwenhuisen and Mark Overmars are with the Institute of Information and Computing Sciences, Utrecht University, The Netherlands (e-mail: {dennis, markov}@cs.uu.nl).

Jur van den Berg is with the Department of Computer Science, University of North Carolina at Chapel Hill, USA (e-mail: berg@cs.unc.edu).

of the algorithm. Finally we report experiments that show the effectiveness of our approach.

II. ROBUST ROADMAPS

The path planning problem is defined as finding a path for a robot R from one configuration to another while avoiding obstacles. Often these obstacles are assumed to be stationary, e.g. walls. The PRM method creates a roadmap G that represents the free space such that paths can be extracted quickly. A consequence of this approach is that the environment is not allowed to change between the moment the roadmap is created and the moment of the query. However, in realistic environments often obstacles are not stationary but can change their position between the moment the roadmap is created and the moment a path is extracted. For example someone can close a door or move a chair. In our algorithm we use the fact that in many applications the environment contains information about which changes in the placements of the obstacles are possible. For example, we usually know where the doors are that can be opened, and in which states they can be (anywhere between open and closed). We also know that a chair is located "somewhere" in a room. We refer to obstacles for which the set of *potential* placements is known a priori as *moving obstacles*. These include obstacles that can be removed from (and re-added to) an environment. So, besides stationary obstacles, a changing environment contains k moving obstacles $M_1, M_2, \ldots, M_k$.

In the remainder of this section, we will briefly restate the algorithm described in [1]. For each moving obstacle M_i a set of potential placements $P(M_i)$ is defined. Each set $P(M_i)$ is partitioned into a finite set of *chunks* $\Delta(M_i) = \{\delta_i^1, \delta_i^2, \ldots\}$. A chunk $\delta \in \Delta(M_i)$ is a subset of the placements of obstacle $M_i : \delta \subset P(M_i)$. The definition of $\Delta(M_i)$ as a partition of $P(M)$ implies the following: $\bigcup \Delta(M_i) = P(M_i)$ and $\delta_i^p \cap \delta_i^q = \emptyset$ when $p \neq q$. The set of all combinations of the positions of the moving obstacles is $D = \Delta(M_1) \times \Delta(M_2) \times \cdots \times \Delta(M_k)$. An element d of D, is denoted by $d = (\delta_1^{q_1}, \delta_2^{q_2}, \ldots, \delta_k^{q_k})$, where the q's are indices of individual obstacle chunks. Such a *composite chunk* d simply states for each moving obstacle in which chunk it is present. We will now describe the algorithm to create robust roadmaps, roughly following the PRM algorithm. For this we need some definitions (taken from [1]).

Definition II.1 (Collision-free)**.** *For a configuration c of the robot and a composite chunk $d \in D$, we define the function $CF(c, d)$ to be true iff c neither collides with the stationary obstacles nor with any of the chunks in d. Analogously, for two configurations c and c', the function $CF(c, c', d)$ is true iff the path generated by the local planner between c and c' neither collides with the stationary obstacles nor with any of the chunks of d.*

To create the roadmap G, a random configuration c in C-space is generated and added to G as a vertex if $\exists_{d \in D} \mid CF(c, d)$. Next, connections (edges) between the vertices need to be added. For this the notion of roadmap path existence is defined.

Definition II.2 (Path existence)**.** *For two vertices c_i and c_j in G and a composite chunk $d \in D$, we define the function $PE(c_i, c_j, G, d)$ to be true iff a (collision-free) path exists in G connecting c_i and c_j if the obstacles are placed according to composite chunk d.*

Only symmetric motions are allowed, thus $PE(c_i, c_j, G, d) = PE(c_j, c_i, G, d)$. In changing environments, *cycles* in the roadmap G are important in providing alternative routes if a path is blocked by a moving obstacle. Adding every potential edge to G (thus creating many cycles) results in many alternative routes but is computationally expensive and makes G unnecessary large which influences the speed of the query phase. Therefore in [1] the notion of *labeled components* is introduced. Labeled components consist of groups of vertices that are collision free for the same subset of D and are also connected for that same subset.

Definition II.3 (Labeled component)**.** *Two vertices $c_i, c_j \in G$ belong to the same* labeled component *L iff*

$$\begin{aligned}\{d \in D \mid CF(c_i, d)\} &= \{d \in D \mid CF(c_j, d)\} \\ &= \{d \in D \mid PE(c_i, c_j, G, d)\}.\end{aligned}$$

Note that vertices that belong to a different labeled component can still be connected by a path in G. Between each pair of labeled components the subset of D for which a path exists is maintained. Such a set is called a *connection set*. An example is shown in Fig. 1.

Definition II.4 (Connection set)**.** *For each pair of labeled components L_i and L_j in G, a* connection set *$CS(L_i, L_j, G) \subset D$ is defined as the set of composite chunks for which a path exists between L_i and L_j:*

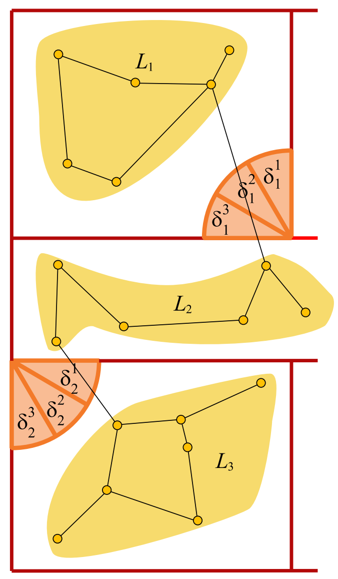
$$CS(L_i, L_j, G) = \{d \in D \mid \exists_{c_i \in L_i, c_j \in L_j}\ PE(c_i, c_j, G, d)\}.$$

The information contained in the connection sets is used to decide when to add an edge to the roadmap G. Only if such an edge connects two different labeled components *and* if the connection set between those labeled components is extended the edge is added. Such an edge is called a *necessary edge*.

Definition II.5 (Necessary edge)**.** *For two vertices $c_i \in L_i$ and $c_j \in L_j$, we define the edge (c_i, c_j) to be* necessary *if it extends the connection set between the two labeled components in G, i.e. $\{d \in D \mid CF(c_i, c_j, d)\} \not\subset CS(L_i, L_j, G)$.*

If a new vertex c has been added to G, a set of neighbor vertices N_c in G is selected. If the edge between c and $c' \in N_c$ is necessary, it is added to G. The generation of random configurations and the addition of necessary edges continues until a predefined stop criterion has been met.

Two more ingredients are needed to create robust roadmaps. If a connection set is extended because a necessary edge is added, it is updated. This update may affect other connection sets as well because new routes through



(a)

$CS(L_1, L_2)$

$$\begin{array}{c|ccc} \delta_2^3 & x & \cdot & \cdot \\ \delta_2^2 & x & \cdot & \cdot \\ \delta_2^1 & x & \cdot & \cdot \\ \hline & \delta_1^1 & \delta_1^2 & \delta_1^3 \end{array}$$

$CS(L_1, L_3)$

$$\begin{array}{c|ccc} \delta_2^3 & x & \cdot & \cdot \\ \delta_2^2 & x & \cdot & \cdot \\ \delta_2^1 & \cdot & \cdot & \cdot \\ \hline & \delta_1^1 & \delta_1^2 & \delta_1^3 \end{array}$$

$CS(L_2, L_3)$

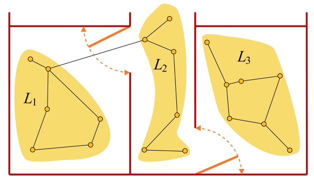
$$\begin{array}{c|ccc} \delta_2^3 & x & x & x \\ \delta_2^2 & x & x & x \\ \delta_2^1 & \cdot & \cdot & \cdot \\ \hline & \delta_1^1 & \delta_1^2 & \delta_1^3 \end{array}$$

(b)

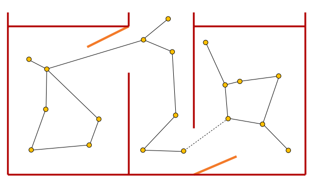
Fig. 1: Connection sets. (a) An environment with two moving obstacles (the two doors). The placements of both doors are partitioned in three chunks. (b) The corresponding connection sets. The connection sets show for all combinations of chunks whether there is a connection or not between the labeled components. An "x" means there is a connection, a dot means there is no connection.

the roadmap may have become available. Therefore the new information needs to be *propagated* to update all other connection sets. An example of propagation is shown in Fig. 2. Also after an update, a connection set may contain every element of D (i.e. a path exists between the two associated labeled components regardless of the placements of the moving obstacles). Because of definition II.3 the two labeled components need to be *merged* and the connection set can be deleted. In [1] it is shown that the procedure described in this section to create roadmaps is probabilistically complete, i.e. if the algorithm is run for a sufficient amount of time, the roadmap is guaranteed to contain a solution for every feasible query.

The roadmap can now be used to extract paths in the query phase. Given the chunks the moving obstacles are in, a path can be quickly generated. If during the execution of the path a moving obstacle changes position, we immediately know if one or more of the edges of the path are invalidated. In that case a new path is created by querying from the current position of the robot to the goal without any additional collision checks.



(a) Three labeled components are present.



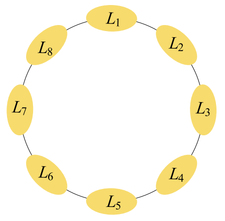
(b) Adding the dotted edge to G not only extends $CS(L_2, L_3, G)$ but also $CS(L_1, L_3, G)$.

Fig. 2: Propagation, the environment consists of two rooms and a hallway separated by doors.

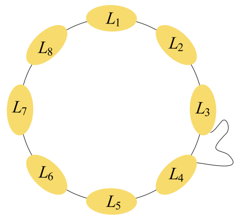
III. DRAWBACKS OF THE ALGORITHM

Although successful in creating robust roadmaps, the algorithm of the previous section has some drawbacks. Since a connection set represents all combinations of chunks for which a path exists in G, the size of the connection sets and thus the amount of time spent on the operations manipulating the connection sets increases exponentially with the number of obstacles and chunks. Suppose that the set of placements $P(M_i)$ of M_i is partitioned in m_i chunks, then the total number of elements in each connection set equals $\prod_{i=1}^{k} m_i$ where k is the total number of moving obstacles. Stated differently, there is an exponential dependency between the number of chunks and the size of the connection sets. For example, if an environment consists of 10 obstacles, each consisting of 4 chunks, a connection set consists of $4^{10} = 1,048,576$ elements. If between every pair of labeled components a connection set is defined, the number of connection sets maintained is equal to the square of the number of labeled components. Note that this number should be divided by 2 since undirected roadmaps are used. Thus if the number of moving obstacles in the algorithm presented in [1] increases, the algorithm quickly becomes computationally infeasible.

A second problem occurring with the propagation algorithm is that if a connection set between two labeled components L_i and L_j is extended, this can potentially affect all other connection sets. Consider the following example shown in Fig. 3: the roadmap G consists of n labeled components $(L_1, L_2, ..., L_n)$. If the labeled components form a closed chain, i.e. labeled component L_i is only neighboring L_{i-1} and L_{i+1} if $1 < i < n$ and L_1 is neighboring L_n, then if a necessary edge is added between L_i and L_{i+1}, this could potentially affect all other n^2 connection sets. In this case the propagation algorithm is relatively expensive. In Fig. 3 it may be necessary to propagate to $n^2 = 8^2$ connection sets.



(a) A closed chain of labeled components each neighboring only two other labeled components.



(b) The connection set $CS(L_3, L_4, G)$ is extended.

Fig. 3: An example in which propagation is expensive.

For many pairs of labeled components (especially those close together) there is only a limited number of moving obstacle placements for which no path exists. The connection sets however store for which combination of placements of the obstacles a path *does* exist. The result is that connection sets are exponentially large while only for a small number of obstacle placement combinations no path might exist. Because we need to perform many operations on these connection sets (check completeness, merge) the running time of the algorithm is for a large part related to these operations.

IV. Improving the Algorithm

Before we propose solutions to the observations of the previous section we translate our problem to the domain of Boolean logic [3]. For this, we use the variables of the chunks as literals (a literal is an atomic formula or its negation in Boolean logic). For example, δ_i^2 means that obstacle M_i is positioned inside chunk δ_i^2. The negation of a chunk, e.g. $\overline{\delta_i^3}$, means that obstacle M_i is not positioned within that chunk. The combination of chunks for which a path is obstructed can now be described using an *obstruction function*.

Definition IV.1 (Obstruction function). *Given two configurations $c_i, c_j \in G$, an* obstruction function *$O(c_i, c_j)$ is a Boolean function that states for which combinations of literals (chunks) the path created by the local planner between c_i and c_j is obstructed. If the values for the literals are known (and thus the positions of the moving obstacles), it evaluates to* TRUE *if the path is obstructed, else it evaluates to* FALSE*. For two labeled components L_i and L_j, the obstruction function $O(L_i, L_j, G)$ states for which combinations of literals no path exist in G between the two labeled components.*

Suppose the local planner reports intersections on the path between configurations c_i and c_j in Fig. 4 with the following chunks: δ_1^3, δ_2^1 and δ_2^2, then the obstruction function for the corresponding edge is: $O(c_i, c_j) = (\delta_1^3 \vee \delta_2^1 \vee \delta_2^2)$. Given the actual placements of the moving obstacles, we know which literals are TRUE and which ones are FALSE. If an obstruction function of an edge evaluates to TRUE then the edge collides for the given placements of the obstacles. Note that the obstruction function of a single edge always consists of only one clause in disjunctive normal form.

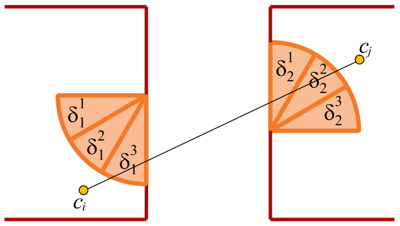


Fig. 4: An example of an edge (c_i, c_j) intersecting multiple chunks. The corresponding obstruction function is $(\delta_1^3 \vee \delta_2^1 \vee \delta_2^2)$.

Using the literals we can also describe the connection sets that state for which composite chunks a path exists between two labeled components. However, considering that between two labeled components there is often only a limited amount of chunks that can block the path between them, we will switch from maintaining connections to maintaining obstructions. Obstructions between labeled components will be maintained in obstruction functions. For each pair of labeled components L_i and L_j, an obstruction function $O(L_i, L_j, G)$ is defined.

Suppose we have two labeled components L_i and L_j. The corresponding obstruction function $O(L_i, L_j, G)$ is initialized with TRUE. This signifies that no path exists for any combination of the literals between L_i and L_j. If an edge is added between $c_i \in L_i$ and $c_j \in L_j$ then the corresponding obstruction function is extended by taking the conjunction of the obstruction function of the edge and the current obstruction function between the labeled components: $O(L_i, L_j, G)$ becomes $O(L_i, L_j, G) \wedge O(c_i, c_j)$. It is also possible that no direct edge is added between two labeled components but connectivity is extended via other labeled components using propagation (see IV-A for more details). Such an example is shown in Fig. 5. Here an obstruction function $O(L_1, L_3, G)$ exists for labeled components L_1 and L_3. Next, an edge between L_2 and L_3 is added to G. Since a path through G is now possible from L_1 to L_3 via L_2, the obstruction function $O(L_1, L_3, G)$ is extended.

Given the values of the literals (e.g. the actual chunks in which the obstacles are) at query time, an obstruction function can be evaluated to check if a path exists between two labeled components for a given set of placements of the obstacles. If it evaluates to FALSE, a connection between the labeled components exists.

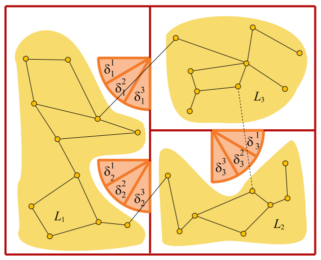


Fig. 5: A more complex obstruction function. Before the dotted edge is added to the roadmap, $O(L_1, L_3, G) = (\delta_1^2 \vee \delta_1^3)$. After the addition $O(L_1, L_3, G) = (\delta_1^2 \vee \delta_1^3) \wedge (\delta_3^1 \vee \delta_3^2 \vee \delta_2^3)$.

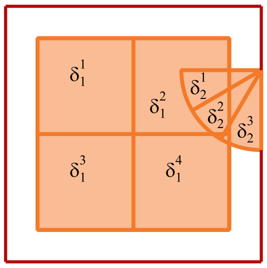


Fig. 6: An environment consisting of two moving obstacles: a door (3 chunks) and a chair (4 chunks).

A. Implementing the Operations

We will now describe how the different operations of [1] can be implemented using obstruction functions. To check if the edge between vertices $c_i \in L_i$ and $c_j \in L_j$ is necessary, we need to verify that the conjunction of $O(c_i, c_j)$ and the current obstruction function $O(L_i, L_j, G)$ yields more valid paths between the labeled components than the current obstruction function $O(L_i, L_j, G)$. This is the case if $O(c_i, c_j)$ evaluates to FALSE (i.e. a path exists) for certain combinations of literals while $O(L_i, L_j, G)$ evaluates to TRUE. Stated differently the new edge is necessary if $O(L_i, L_j, G)$ is *not* an implication of $O(c_i, c_j)$. To check for a subset relation we create the function $(\overline{O(L_i, L_j, G)} \vee O(c_i, c_j))$. If this is a *tautology* (i.e. true for every combination of literals), then the edge is not necessary. Checking whether such a function is a tautology can be an expensive process because of the large number of combinations of literals.

To speed up the evaluation of the obstruction functions, we will transform them to a satisfiability test. Such a test checks whether a combination of literals exists that makes a function TRUE. If this is not possible the test outputs *unsatisfiable*. Testing whether a function is a tautology is equivalent to testing if its negation is unsatisfiable. Now we can restate the definition of a necessary edge:

Definition IV.2 (Necessary). *For two vertices $c_i \in L_i$ and $c_j \in L_j$ and obstruction function $O(L_i, L_j, G)$, we define the edge (c_i, c_j) having obstruction function $O(c_i, c_j)$ to be* necessary *if the following function is unsatisfiable:*

$$\overline{(\overline{O(L_i, L_j, G)} \vee O(c_i, c_j))}$$

which simplifies to

$$(O(L_i, L_j, G) \wedge \overline{O(c_i, c_j)}).$$

The satisfiability problem is well-known in Boolean logic and is NP-complete [4], [6]. Fortunately many high performance heuristics have been developed, see e.g. the proceedings of the International conference on Theory and Applications of Satisfiability Testing [2].

Recall that after updating a connection set the new information was propagated to the other connection sets by recursively updating all neighboring labeled components. Because we now use obstruction functions, the propagation algorithm needs to be adapted and the new necessary edge test has to be used. It is shown as Alg. IV.1. In the initial call, O_n is the obstruction function of the newly added necessary edge.

Algorithm IV.1 PROPAGATE (L_i, L_j, G, O_n)

```
1: $O(L_i, L_j, G) \leftarrow O(L_i, L_j, G) \wedge O_n$
2: for all neighbors $L_k$ of $L_i$ do
3:     $O_n \leftarrow O(L_i, L_j, G) \vee O(L_i, L_k, G)$
4:     if $(O(L_j, L_k, G) \wedge \overline{O_n})$ is unsatisfiable then
5:         PROPAGATE $(L_j, L_k, O_n)$
6: for all neighbors $L_k$ of $L_j$ do
7:     $O_n \leftarrow O(L_i, L_j, G) \vee O(L_j, L_k, G)$
8:     if $(O(L_i, L_k, G) \wedge \overline{O_n})$ is unsatisfiable then
9:         PROPAGATE $(L_i, L_k, O_n)$
```

After propagation, labeled components that are connected for every combination of chunks need to be merged. In principle the procedure as described in [1] can be used. There, two labeled components were merged if their connection set was *complete*. Because here we use obstruction functions, we need to redefine completeness.

Definition IV.3 (Complete). *The obstruction function $O(L_i, L_j, G)$ is* complete *iff for every combination of placements of the obstacles $O(L_i, L_j, G) =$ FALSE.*

Stated differently, to check if an obstruction function is complete we need to check whether it is *unsatisfiable*.

B. Preliminary Functions

Since multiple chunks are associated with one obstacle, the associated literals are not independent. Besides the information that is contained within the obstruction functions, there is also a list of assumptions that we can add to those functions. Given moving obstacle M_i, partitioned in m_i chunks $\Delta(M_i) = \{\delta_i^1, \delta_i^2, \dots, \delta_i^{m_i}\}$, we know the following two facts:

1) Moving obstacle M_i will be present in one of the chunks, thus $\bigvee_{p=1}^{m_i} \delta_i^p =$ TRUE.
2) An obstacle will never be in two chunks at the same time. Therefore $\overline{\delta_i^p \wedge \delta_i^q} =$ TRUE for all $p \neq q$.

The above two rules apply to all k moving obstacles $M_1, M_2, \dots, M_k$. The preliminary functions are always added as assumptions when a function is tested for satisfiability. For example in the environment shown in Fig. 6 we know the following 11 functions to be true: $\delta_1^1 \vee \delta_1^2 \vee \delta_1^3 \vee \delta_1^4$, $\overline{\delta_1^1 \wedge \delta_1^2}$, $\overline{\delta_1^1 \wedge \delta_1^3}$, $\overline{\delta_1^1 \wedge \delta_1^4}$, $\overline{\delta_1^2 \wedge \delta_1^3}$, $\overline{\delta_1^2 \wedge \delta_1^4}$, $\overline{\delta_1^3 \wedge \delta_1^4}$, $\delta_2^1 \vee \delta_2^2 \vee \delta_2^3$, $\overline{\delta_2^1 \wedge \delta_2^2}$, $\overline{\delta_2^1 \wedge \delta_2^3}$ and $\overline{\delta_2^2 \wedge \delta_2^3}$.

C. Limiting the Number of Obstruction Functions

The number of obstruction functions (and in the original algorithm the number of connection sets) is an important factor in the running time of our algorithm. The more data is propagated throughout the roadmap, the more obstruction functions will be created. Because of the Boolean operations during propagation, the obstruction functions also get more complex. However, the larger the *propagation depth*, the further two labeled components are apart in terms of graph distance. If no information is propagated, our algorithm behaves the same as standard PRM. The first level of propagation is between neighboring labeled components. The next level is between labeled components that are connected via a third labeled component etc. To guarantee that no unnecessary edge is added, propagation needs to continue as long as obstruction functions are extended (see Alg. IV.1).

Our ultimate goal is to create roadmaps that represent the connectivity of the environment without getting too dense to provide for a fast query phase. We have conducted experiments to investigate the relation between the maximum propagation depth and the density of the roadmap. The algorithm was run until a certain number of vertices were added to the roadmap. With this number fixed, the number of edges in the roadmap is a good measure of its density. We compared the maximum propagation depth with the number of edges in G. The environment consists of 10 obstacles whose placement sets were all represented by 4 chunks. The results of this experiment are shown in Table IV-C.

Prop. depth	Time (s)	Nr. edges	Nr. obstruction func.
0	6.6	4529	0
1	9.6	867	352
2	17.3	676	706
3	51.2	652	1525
4	133.1	652	1458

TABLE I: Relation between propagation depth and number of edges.

As can be seen from the results, the number of edges decreases quickly when the maximum propagation depth is increased. If the propagation depth gets too large however, the number of edges hardly decreases anymore while the running time increases dramatically. This can be easily explained by realizing that an obstruction function between two labeled components is only used when a direct edge between those labeled components is tested for usefulness. The further labeled components are apart, the smaller the probability this will happen. Therefore we can limit the propagation depth such that only a very small number of useless edges is added to the roadmap while the running time decreases drastically. We establish the maximum propagation depth automatically. In standard PRM connections are usually only tried to vertices within a certain distance, the *maximum neighbor distance*. We use this distance to establish the maximum propagation depth. If two labeled components are further apart than the maximum neighbor distance then propagation stops. The distance between two neighboring labeled components can easily be defined as the distance between their center points where a center point is defined as the point with coordinates that are the average of all configurations in the labeled component. If the two labeled components are not neighboring we use the cumulative distance.

Early in the execution of our algorithm there are many different labeled components consisting of only one or a few vertices. In a later stage, as the number of edges in the roadmap increases, many of these will be merged into larger labeled components. To speed up this process we start the roadmap creation process by *seeding*. By seeding we mean using standard PRM in the free space only (i.e. not colliding with the stationary obstacles nor any of the chunks). This creates labeled components between the chunks of the moving obstacles that can later act as bridges between the other labeled components. The number of vertices that are added in the seeding process is determined automatically. Ideally it should be related to the difficulty of the problem. If a problem consists of many chunks it is considered difficult. Using information gathered during the seeding process we can estimate this difficulty. In the seeding process an edge is rejected if it intersects with a chunk. We keep track of the ratio between accepted and rejected edges and use that to determine the number of vertices used for seeding. (Note that, to determine this ratio, we ignore edges that are rejected because they collide with one of the stationary obstacles.) Using the maximum number of vertices that we use as a stop-criterion for the algorithm, the ratio is used to decide how many vertices are added in the seeding phase.

V. EXPERIMENTS

Our algorithm has been implemented in C^{++} and experiments have been conducted on a Pentium IV 2.4GHz having 1GB of memory. To check the satisfiability of the Boolean functions we used MiniSat [5]. This is a very efficient SAT solver that is publicly available. We have performed a few different experiments. We compared the algorithm of this paper with a standard implementation of PRM (*standard* PRM) and with the algorithm as described in [1] (*straightforward method*). To provide for a fast query phase, we need to know which edges of the roadmap collide with which (placements of) the moving obstacles. This information is inherently available in our algorithm and for a fair comparison it should also be available to the standard PRM. Therefore, during preprocessing we collision check the vertices and edges of the standard PRM algorithm both with the stationary obstacles and the chunks. The potential collisions with one or more chunks are stored within the roadmap such that at query time this information is readily available. To gain insight in the performance gain of seeding we did all experiments both with and without seeding.

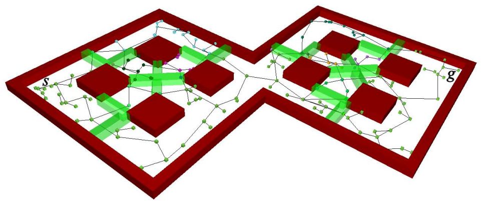


Fig. 7: The Double Puzzle scene. Every “door” can be positioned in two chunks.

For the experiments we used three environments: **Puzzle**, **double puzzle** and **office**. Double puzzle is shown as Fig. 7 and has 8 moving obstacles. For puzzle we simply used half of this environment. The office environment (shown in Fig. 8) represents 5 different rooms around a central staircase. The two top and two bottom rooms can be reached using ordinary doors all represented by 3 chunks. Also between the rooms are doors. Inside the rooms are desks and chairs that can be positioned in 4 different chunks. The left room can be reached by means of a sliding door. Inside this room there are two boxes that can both be placed in 4 different chunks. In total this environment consists of about 12 million different combinations of chunks. Because of the high number of chunks, the office scene was computationally infeasible for the straightforward method.

All experiments ran until a predetermined number of vertices were added to the roadmap. In this way it is easy to compare our algorithm to standard PRM. Also all other parameters were kept constant to provide for a fair comparison; they are shown in Table II. Besides the running time we also report the number of edges in the roadmap since this number relates closely to the query time (since the number of vertices is the same in all algorithms).

	MaxNrVert	MaxNeighbDist	MaxNrNeighb
Puzzle	200	26%	10
Double puzzle	400	49%	10
Office	800	15%	10

TABLE II: Parameter settings. The maximum neighbor distance is a percentage of the diameter of the environment.

Every experiment was repeated 50 times and the results were averaged. They are shown in Table III. As can be seen from the results, in the simple environments our algorithm outperforms standard PRM in terms of time. This can be mainly attributed to the low number of chunks and therefore the relative high cost of collision checking as compared to the time spend in manipulating obstruction functions. Because standard PRM does not have the advantage of labeled components, it is not able to save collision checks and therefore its running time is higher. In the simple puzzle environment the overhead of the satisfiability tests is relatively high. Therefore the straightforward method outperforms the Boolean logic method by a small factor. If the number of chunks gets larger, the method using Boolean logic is faster. This is already the case in the double puzzle environment that has 16 chunks. In the office environment, which is much more complex, the running times of standard PRM and Boolean logic are comparable. In all cases the complexity of the roadmaps was much lower in our algorithm as can be seen from the number of edges resulting in a better query time. Fig. 8 shows examples of the roadmaps generated by standard PRM and our method. The connectivity of the roadmaps is comparable but since in the second roadmap only useful edges were allowed, the roadmap is much sparser. As can be seen from all experiments, seeding helps by lowering the running time while keeping the number of edges equal.

	Puzzle		**Double puzzle**		**Office**	
	t(s)	n	t(s)	n	t(s)	n
Standard PRM	2.03	1652	7.61	3130	5.74	6368
Straightforward	0.54	207	4.03	430		
Boolean logic	0.79	207	2.42	410	5.20	902
w/o seeding	1.22	218	3.52	424	9.32	981

TABLE III: Results averaged over 50 runs. **t(s)** is running time in seconds, **n** is number of edges.

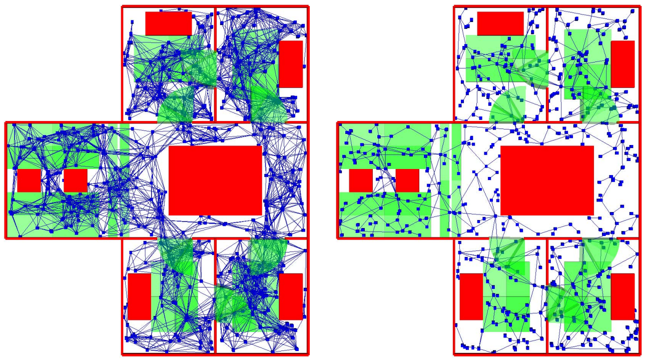

(a) Standard PRM. **(b)** Our algorithm.

Fig. 8: Example roadmaps in the office environment. Chunks are shown in slightly different shades.

VI. CONCLUDING REMARKS

In this paper we have improved the basic algorithm presented in [1] for motion planning in changing environments. We first pinpointed its bottlenecks and then suggested improvements. A first problem was the memory consuming connection sets that kept path existence information for every combination of chunks. If the environments become more complex the memory consumption becomes too high for this approach. Therefore we have switched from maintaining full connection information to maintaining only collision information. While this makes no difference from a theoretical perspective, in practice it turns out to be a major gain. Secondly, the operations concerning the connection sets were rather brute force. To speed up these operations, we translated the problem to the domain of Boolean logic. While the problem of checking the completeness of an obstruction function still remains difficult, within this domain many very fast heuristics have been created. To determine if two labeled components can be merged, we need to check whether the related obstruction function is complete. This problem was translated to a satisfiability test that could be performed quite efficiently in practice.

We have also presented two heuristics. The first heuristic limits the propagation depth. Because of this limitation, information in the obstruction functions is not complete anymore and thus unnecessary edges may be added to the roadmap. In our experiments we showed that the number of unnecessary edges was very low while the gain in performance was huge. We also described a method to automatically determine the propagation depth without introducing additional parameters. The second heuristic speeds up the roadmap creation process by first adding some vertices in the free space. This heuristic prevents that in the early phases of the algorithm many small labeled components are created that later need to be merged.

We conducted experiments that showed that our algorithm performs efficiently in realistic environments. Compared to standard PRM, it creates much sparser roadmaps (while maintaining the same connectivity) that provide for fast queries and re-planning.

REFERENCES

[1] J. P. van den Berg, D. Nieuwenhuisen, L. Jaillet, and M. H. Overmars. Creating robust roadmaps for motion planning in changing environments. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2415–2421, 2005.

[2] A. Biere and C.P. Gomes, editors. *Theory and Applications of Satisfiability Testing - SAT 2006, 9th International Conference, Seattle, WA, USA, August 12-15, 2006, Proceedings*, volume 4121 of *Lecture Notes in Computer Science*. Springer, 2006.

[3] G. Boole. The calculus of logic. *Cambridge and Dublin Mathematical Journal*, III (1848):183–198, 1848.

[4] S. Cook. The complexity of theorem proving procedures. In *ACM Symposium on Theory of Computing*, pages 151–158, 1971.

[5] N. Eén and N. Sörensson. Minisat 1.14, 2005.

[6] M.R. Garey and D.S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W.H. Freeman & Co., 1979.

[7] L. Jaillet and T. Siméon. A prm-based motion planner for dynamically changing environments. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1606–1611, 2004.

[8] A. Kamphuis and M.H. Overmars. Finding paths for coherent groups using clearance. In *IEEE International Conference on Robotics and Automation*, pages 3815–3822, 2004.

[9] L.E. Kavraki, P. Švestka, J.-C. Latombe, and M.H. Overmars. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *IEEE Transactions on Robotics and Automation*, 12:566–580, 1996.

[10] 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, 2001.

[11] P. Leven and S. Hutchinson. A framework for real-time path planning in changing environments. *International Journal of Robotics Research*, 21(12):999–1030, 2002.

[12] D. Nieuwenhuisen, A. Kamphuis, M. Mooijekind, and M.H. Overmars. Automatic construction of roadmaps for path planning in games. In *International Conference on Computer Games: Artificial Intelligence, Design and Education*, pages 285–292, 2004.

[13] G. Song and N.M. Amato. Using motion planning to study protein folding pathways. *Journal of Computational Biology*, 9(2):149–168, 2001.