Roadmap-based Motion Planning in Dynamic Environments

Jur P. van den Berg

Mark H. Overmars

Roadmap-based Motion Planning in Dynamic Environments

Jur P. van den Berg   Mark H. Overmars

April 2004

Abstract

In this paper a new method is presented for motion planning in dynamic environments, that is, finding a trajectory for a robot in a scene consisting of both static and dynamic, moving obstacles. We propose a practical algorithm based on a roadmap that is created for the static part of the scene. On this roadmap an approximate time-optimal trajectory from a start to a goal configuration is computed, such that the robot does not collide with any moving obstacle. The trajectory is found by performing a search for a shortest path on an implicit grid in state-time space. The approach is applicable to any robot type in configuration spaces with any dimension, and the motions of the dynamic obstacles are unconstrained, as long as they are known beforehand. The approach has been implemented for a free-flying robot in a three-dimensional workspace and experiments show that the method achieves interactive performance in complex environments.

1   Introduction

Motion planning is of great importance, not only in robotics, but also in other fields such as virtual environments, maintenance planning and computer-aided design. Much research has been done on motion planning in static environments and both exact and approximate methods have been devised [9]. A popular approximate method is the probabilistic roadmap planner (PRM) [7, 13]. It is a generic method that creates a roadmap in a preprocessing phase that represents the connectivity of the free configuration space. Individual motion planning problems can then be solved quickly by finding a path in the roadmap. The method has successfully been used in high-dimensional configuration spaces of complex environments.

The extension of the motion planning problem to dynamic environments has been extensively studied as well [1, 2, 3, 4, 5, 6, 8, 12], but only a limited number of practical algorithms have been devised that deal genericly with moving obstacles. PRM could be extended to the dynamic motion planning problem by incorporating the absolute notion of time as an additional dimension in the configuration space. However, since the obstacle motions are not assumed to be periodic (cyclic), the configuration space is highly transitory. As a consequence, building a roadmap during a preprocessing phase is not useful for such configuration spaces. Therefore, single-shot variants of PRM [10] have been the methods of choice for this type of problems [1, 6, 8]. In such methods a roadmap is built incrementally in the form of a directed tree oriented along the time axis for each planning query. Although some promising results have been achieved in real world situations, these methods are less suitable in large scenes in which besides dynamic obstacles, a large number of static obstacles is present. This is because all the effort has to be done in the query phase, which undermines the often required real-time performance of the method.
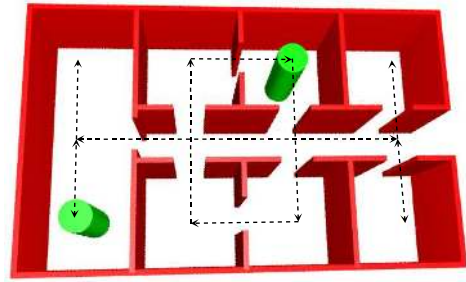
Figure 1: A dynamic environment with two moving obstacles (cylinders). The cylinders move cyclically along the dotted lines.

In this paper, we propose a new method which is based on a roadmap built in a pre-processing phase. The roadmap is built on the static part of the scene without the dynamic obstacles and without the additional dimension for time. This can be done using a standard PRM method, but devising a roadmap on the drawing table may suffice just as well. In the query phase, only the dynamic obstacles are dealt with.

Our method searches for a near-time-optimal trajectory between a start and a goal configuration in the roadmap, without collisions with the dynamic obstacles. The trajectory is found by performing a search for a shortest path on an implicit grid in state-time space. This approach is also used in [3] to find a trajectory avoiding the dynamic obstacles on a *path* that is collision-free with respect to the static obstacles. We extend this approach to a roadmap, which considerably enlarges the maneuverability of the robot and hence the chance that a trajectory is found.

Our method follows the same principles as a theoretical method of Fujimura [5], which computes an exact time-optimal path in a roadmap using visibility graphs in state-time space. His method though works only for point robots in two-dimensional environments where the dynamic obstacles are constrained to piecewise linear motions without rotation.

By choosing an approximate approach, we were able to lift these drawbacks. Our method is practical and applicable to any robot type in configuration spaces with any dimension. The only ingredient the method requires is a roadmap for the robot amidst the static obstacles. The shape and motions of the dynamic obstacles are completely free: they may move with any speed following any trajectory, may deform and even jump (warp), as long as the motions are known beforehand. That is, given a position of the robot at a time t we must be able to answer the question whether the robot is collision-free. As in [5] we do not put constraints on the robot's motion, except for an upper bound on its velocity.

The method has been implemented for free-flying robots with six degrees of freedom, and experiments show that it achieves interactive performance in confined dynamic environments (see Fig. 1).

The rest of the paper is organized as follows. A formal definition of the problem is given in section 2. In section 3 we describe the global approach to our method. The problem is split up in two parts: finding local trajectories on single arcs of the roadmap and finding a global trajectory through the entire roadmap. These will be discussed in sections 4 and 5 respectively. In section 6 some extensions and optimizations to the algorithm are discussed and section 7 describes the experimental results.

3

2 Problem description

2.1 State-time space

The static motion planning problem is generally formulated in terms of the *configuration space* C, the set of all possible configurations of the robot. The dimension of the configuration space corresponds to the number of the robot's degrees of freedom. C_{free} denotes the subset of C containing all *collision-free* configurations of the robot. The motion planning problem is now defined as finding a curve from a start configuration to a goal configuration that is entirely contained in C_{free}, possibly satisfying some additional robot-specific constraints.

To extend the definition to motion planning in dynamic environments, an absolute notion of time is incorporated in C. To be consistent with previous literature on the topic, we call this resulting space the *state-time space* [3]. It consists of pairs (x, t), where x is an element of C denoting the robot's state, and t a scalar denoting the time. The robot is represented by a point in state-time space, and both static and dynamic obstacles in the workspace transform to static obstacles in state-time space. We call them state-time obstacles.

Finding a collision-free path in the state-time space is not enough to solve the problem, for the robot is subjected to constraints on its motion. Also, it cannot go back in time. To accentuate this difference, a path obeying the dynamic constraints is called a *trajectory*.

2.2 The roadmap

The roadmap consists of a set of nodes and a set of arcs in the configuration space that must be collision-free with respect to the *static* obstacles. Hence, the roadmap can be constructed in a preprocessing phase. The start and goal configurations are assumed to be present in the roadmap as nodes. If not, they can be connected to the roadmap in the query phase. Our method is applicable to both directed and undirected roadmaps, but for now we assume the arcs to be undirected. In section 6 we discuss how the method is adapted to work with directed roadmaps as well.

The idea of using a preprocessed roadmap is that during the query phase, the static obstacles do not need to be considered in collision checks, which saves a large amount of time. Actually, in the rest of the paper we can simply ignore the static obstacles. Also, the search space for feasible trajectories is substantially reduced; the configuration space is basically brought down to a one-dimensional structure, which makes the problem tractable. If the roadmap given is well covering the free part of the static configuration space, this reduction should hardly affect the chance that a trajectory is found.

The use of a roadmap may have practical advantages as well. In many real-world environments, such as factory floors, sea- and airports, etc., the autonomic robots present are constrained to move along prespecified networks of paths (for instance along lines painted on the floor). They can be modeled perfectly into a roadmap [5].

The quality of the trajectory computed by our method depends directly on the quality of the roadmap. Therefore, using a roadmap containing smooth, natural paths is preferred. The creation of roadmaps is not the topic of study in this paper. Many techniques exist for this, for example the PRM approach. To have a choice of alternative paths it is though important that the roadmap contains cycles (see e.g. [11]).
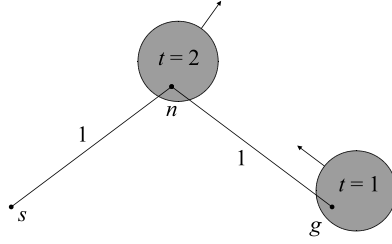
Figure 2: A small example roadmap with a moving obstacle (gray disc). A robot moving from s to g has to wait until the moving obstacle has cleared n.

2.3 The problem

The problem we want to solve is the following. Let R be a robot with an upper bound $v_{\max}$ on its velocity in a two- or three-dimensional workspace containing both static and dynamic obstacles, and let us be given a roadmap for R that is collision-free with respect to the static obstacles in the scene. Let s and g be the start and goal node in the roadmap respectively, and let t_0 be the start time. Then, the problem is to compute a feasible trajectory on the roadmap for the robot R starting at s at t_0 and reaching g as quickly as possible without collisions with moving obstacles.

To give maximal flexibility to the dynamic obstacles, the only way in which the state-time space is sensed is by means of a boolean function $cf(x, t)$ that, given a state $x \in C$ and a moment in time t, reports whether the robot configured at x collides with any moving obstacle at time t.

3 Global approach

To find a trajectory from s to g, a straightforward Dijkstra-search in the roadmap is not possible, since it is not always best to arrive as early as possible on each of the nodes. We illustrate this using a simple example roadmap consisting of three nodes s, n and g with arcs between s and n, and n and g (see Fig. 2). It takes one unit of time for the robot to traverse each arc. An obstacle is moving from g at $t = 1$ to n at $t = 2$ and then moves away from the roadmap. If a robot starts at s at $t = 0$, it is able to reach n at $t = 1$, but then it is not possible to reach g, because the path is blocked by the moving obstacle. If the robot would wait somewhere on the arc (s, n) and arrive at n at $t > 2$, the path to g is free. It would, however, not be useful to arrive even later at n, because the robot would then arrive in the same *free interval* on n. A free interval on a node n is defined as follows:

Definition 3.1. *A free interval on a node n is a maximal continuous segment in time in which the robot configured at n is collision-free.*

It is easy to see that it is not useful to arrive later in the same free interval. A trajectory arriving early at the interval can wait on the node for the rest of the free interval, so arriving later in the same interval does not extend the possibilities of reaching g. This observation is crucial for the method presented. In fact, the problem can be expressed fully in terms of the free intervals on the nodes by modeling each free interval on a node as a vertex in an implicit (directed) graph, which we will call the *interval graph*. Edges exist in the graph between

two vertices when there is an arc between the corresponding nodes in the roadmap and an appropriate trajectory exists between the associated intervals. Then, a trajectory between the start and goal node can be found in this interval graph.

Our approach searches the interval graph using a modified A*-search. We do not compute the interval graph explicitly, but in a lazy fashion by sending so called *probes* through the roadmap in search for a trajectory toward the goal node. We will describe our algorithm in two stages. The first deals with computing a feasible *local* trajectory on a single arc in the roadmap, i.e. it controls the behavior of a single probe. In the second stage we discuss the overall interval graph search and global probe management to find a *global* trajectory.

In the above example only normal local trajectories were considered, i.e. trajectories that originate at one node of an arc and advance to the other node of the arc. However, a second type of local trajectory has to be taken into account as well: trajectories returning to a later free interval on the same node they originate from. They first move away from the node along an arc to make room for a moving obstacle after which they return to the node.

So in the search for a global trajectory toward the goal node, two types of local trajectories must be considered; those that move to the other end of the arc and those that return to the same node. To distinguish between them they are called *advancing* and *returning* trajectories respectively.

4   Local trajectories

In this section we discuss how a near-time-optimal local trajectory is computed along a single arc of the roadmap. We follow an approach similar to [3] by discretizing the state-time space into a grid. The only dynamic constraint the robot is subjected to is a bound $v_{\max}$ on its maximum velocity.

We assume that the roadmap arcs are undirected. For conceptual clarity though, they are considered as two separate 'directed' arcs in the rest of this paper, such that each arc has its own *destination node*.

4.1   The state-time grid

Since we consider trajectories along an arc of the roadmap, a configuration of the robot is reduced to a single variable representing the distance traveled along the arc. Slightly abusing the notation, we denote this variable by x. The resulting state-time space is two-dimensional and consists of pairs (x, t). Obstacles in state-time space may have any shape, since we do not constrain their motions.

We discretize the state-time space by choosing a small time step τ and a principal velocity v_p within the velocity bound $v_{\max}$. The actual velocity v is constrained to be either v_p, 0 or $-v_p$ and may only change at given times $k\tau$, where k is an integer. Given a state-time (x, t), new state-times of the robot are calculated using the following equation of motion:

$$v \in \{v_p, 0, -v_p\}$$

$$x(t + \tau) = x(t) + v\tau$$

This results in a regular two-dimensional grid of state-times (i.e. points in state-time space) in which the robot can be. The spacings in the grid are τ along the time axis and $v_p\tau$ along the state axis (see Fig. 3). Let l be the length of the arc. We choose v_p to have the
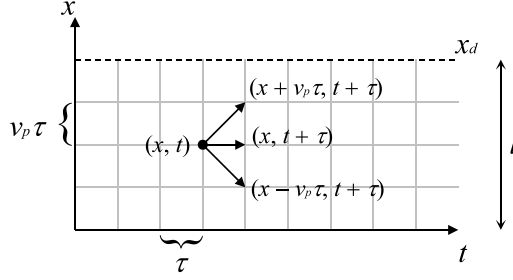
Figure 3: A state-time grid of a roadmap arc. It shows which neighbors are reachable from a given state-time (x, t).

largest value smaller than $v_{\max}$ such that $l/(v_p\tau)$ is an integer, i.e. that the destination node of the arc can be reached exactly in an integer number of time steps. The grid is bounded along the state axis by the length of the arc.

From a given state-time (x, t), three other state-times are reachable, each one associated with a different choice for the velocity. These are $(x + v_p\tau, t + \tau)$, $(x, t + \tau)$ and $(x - v_p\tau, t + \tau)$ (see Fig. 3). This defines a directed graph on the state-time grid, in which the local trajectories can be found. Note that neither the grid nor the graph are explicitly constructed.

4.2 Finding a local trajectory

The problem of finding a local trajectory is defined as follows. Given an arc a and a source state-time (x_s, t_s) on a, find a path in the grid to the first reachable free interval at the destination node of a. The destination node is denoted by x_d. In case of a returning trajectory x_s equals x_d. To prevent that the algorithm immediately returns success in this case, we state that x_d must be reached in an *unvisited* free interval.

A near-time-optimal trajectory can be found by finding a shortest path from (x_s, t_s) to x_d in the directed graph defined on the state-time grid. In [3] an A*-algorithm is used to find the shortest path, but in our case it can be implemented more efficiently using a stack; this requires less collision checks and the elementary operations on the datastructure are cheaper.

The algorithm is initialized with the source state-time (x_s, t_s) on the stack. In every loop the top element (x, t) is popped from the stack. If the corresponding state-time has not been visited before and if it is collision-free with respect to the moving obstacles, the reachable grid points $(x - v_p\tau, t + \tau)$, $(x, t + \tau)$ and $(x + v_p\tau, t + \tau)$ are pushed onto the stack *in this particular order* (see Alg. 1). This means that the most promising step (advancing toward x_d) is considered first. The algorithm runs until the stack is empty or the destination state x_d has been reached in an unvisited free interval (see Alg. 2). Backpointers and information about whether a state-time has been visited before are maintained.

We assume that the time step τ is chosen small enough such that collision checking each of the neighboring state-times is enough to determine whether the trajectory between them is collision-free. Such an approximation is also done in PRM when the arcs of the roadmap are checked for collisions [13].

Fig. 4 shows the working of the algorithm in an example state-time grid. It is easy to see that the algorithm indeed yields a trajectory arriving as early as possible on the destination node.

7

Algorithm 1 DoStep()
___
1: $(x,t) \leftarrow$ StackPop()
2: **if not** (x,t).visited **and** $cf(x,t)$ **then**
3:     StackPush$(x - v_p\tau, t + \tau)$
4:     $(x - v_p\tau, t + \tau)$.backpointer $\leftarrow (x,t)$
5:     StackPush$(x, t + \tau)$
6:     $(x, t + \tau)$.backpointer $\leftarrow (x,t)$
7:     StackPush$(x + v_p\tau, t + \tau)$
8:     $(x + v_p\tau, t + \tau)$.backpointer $\leftarrow (x,t)$
9:     (x,t).visited $\leftarrow$ true
10:     **return** (x,t)
11: **else**
12:     (x,t).visited $\leftarrow$ true
13:     **return** NULL
___

Algorithm 2 FindLocalTrajectory(x_s, t_s, x_d)
___
1: StackPush(x_s, t_s)
2: **repeat**
3:     $(x,t) \leftarrow$ DoStep()
4: **until** $(x = x_d$ **and** the interval on x_d at time t is unvisited) **or** StackEmpty()
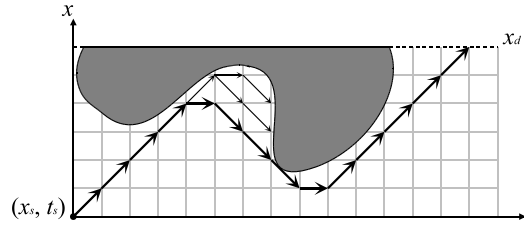___

Figure 4: Finding a trajectory from (x_s, t_s) to x_d. The thin arrows indicate the space explored by the algorithm and the thick arrows form the resulting trajectory. The gray object is a state-time obstacle.

For the correctness of the algorithm we need to prove that we reach x_d at the first possible moment in time. Let t_f be the time at which the destination node x_d is first reachable from a given source state-time (x_s, t_s). Hence, there exists at least one trajectory between (x_s, t_s) and (x_d, t_f) in the grid. Let $\mathcal{T}$ be the set of all trajectories between (x_s, t_s) and (x_d, t_f). We define one of them to be the *highest*, that is the trajectory T_h, which is defined as follows: $T_h(t) = \max_{T \in \mathcal{T}} T(t)$, for $t_s \leq t \leq t_f$, where $T(t)$ is the state of trajectory T at time t. It is easy to see that T_h is itself an element of $\mathcal{T}$.

Theorem 4.1. *Given a source state-time (x_s, t_s) and a destination node x_d that is earliest reachable at time t_f, the above algorithm finds the highest trajectory T_h between (x_s, t_s) and (x_d, t_f).*

Proof We will prove that from every state-time (x, t) on T_h, the algorithm proceeds to the successor of (x, t) on T_h. Since the source state-time (x_s, t_s) also lies on T_h, the algorithm will then find a trajectory from (x_s, t_s) to (x_d, t_f) that exactly equals T_h.

Suppose the algorithm has proceeded a number of steps along trajectory T_h up to some state-time (x, t) on T_h. At this point T_h has three possible successors:

- The successor on T_h is $(x + v_p\tau, t + \tau)$. In this case the algorithm will follow T_h, since it is the most promising step.

- The successor on T_h is $(x, t + \tau)$. The algorithm at this point first proceeds to state-time $(x + v_p\tau, t + \tau)$. Suppose the algorithm finds a trajectory to x_d from this state-time. Then, this trajectory can not reach x_d before t_f, because t_f is the first time at which x_d is reachable. So, at some time this trajectory reaches again a state-time on T_h, but this is also not possible, because then T_h would not be the highest trajectory to (x_d, t_f). Hence, no trajectory is found at all from $(x + v_p\tau, t + \tau)$. So, eventually the algorithm returns to state-time (x, t) and now evaluates $(x, t + \tau)$, which is also the successor of (x, t) on T_h.

- The successor on T_h is $(x - v_p\tau, t + \tau)$. The algorithm now first proceeds to $(x + v_p\tau, t + \tau)$, and then to $(x, t + \tau)$, but for the same reason as above it will not find a trajectory in either of these cases. So eventually the algorithm evaluates $(x - v_p\tau, t + \tau)$, which is also the successor of (x, t) on T_h.

Hence, for every state-time on T_h, which includes (x_s, t_s), the algorithm follows the trajectory T_h, so the trajectory the algorithm will find exactly equals T_h. ∎

In the example of Fig. 4 an advancing trajectory is shown, but the method works equally well for returning trajectories (see Fig. 5). The algorithm does not immediately return success, because it starts in an already visited interval. Hence, the algorithm proceeds until an unvisited interval has been reached. How to determine whether an interval has been visited is discussed in section 5.4.

The algorithm described above finds a trajectory to the first reachable free interval on the destination node. However, in the example of section 3 we saw that the destination node had to be reached in the second reachable free interval. The algorithm is easily adapted to find trajectories to next intervals as well. If the search is not terminated when the first reachable free interval is found (line 4 of Alg. 2), a trajectory to next intervals will be found as well (see Fig. 6).
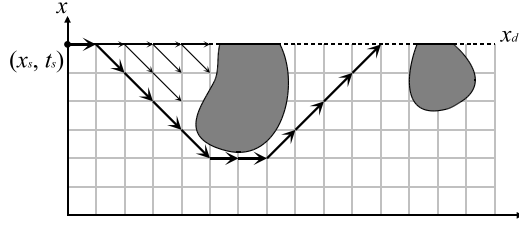
Figure 5: Finding a returning trajectory originating at (x_s, t_s) and arriving at x_d in an unvisited free interval.
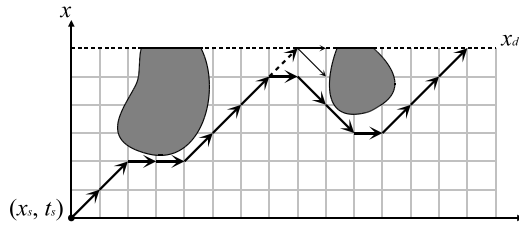

Figure 6: Finding a trajectory to the second reachable interval on x_d. The dotted arrow indicates the trajectory to the first reachable interval.

As proved above, Alg. 2 finds a shortest path in the graph defined upon the grid. The associated trajectory is *near*-optimal when abstracting from the grid, but as the time step τ approaches zero, the trajectory approaches the continuous time-optimal trajectory. For smaller τ the algorithm obviously becomes slower, so the choice of τ gives a trade-off between accuracy and speed.

5 Global trajectories

In the previous section we discussed how to compute a local trajectory on a single arc. In this section we will show how the global trajectory from a start node s to a goal node g through the roadmap is found. As with the local trajectories, the algorithm will find a near-time-optimal trajectory.

5.1 The interval graph

In section 3 we already gave a short introduction about the implicit directed interval graph. Each vertex i in the interval graph corresponds to a free interval on a node in the roadmap. The node associated with an interval vertex i is denoted $n(i)$. An important notion is the time at which the interval is *first reachable* for the robot (see Fig. 7). We denote this time by $t_f(i)$.

An edge exists in the interval graph from a vertex i_j to a vertex i_k when both the following conditions hold:
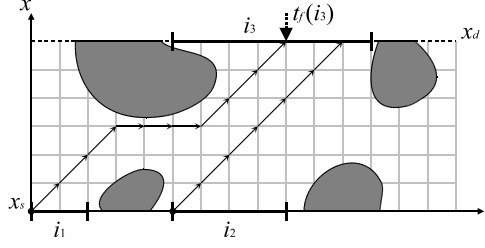
Figure 7: A state-time grid showing free intervals on the nodes and trajectories between them. An edge from i_2 to i_3 is not admitted in the interval graph.

- Their associated nodes $n(i_j)$ and $n(i_k)$ are either connected by an arc in the roadmap, or $n(i_j) = n(i_k)$.

- There exists a local trajectory starting at time $t_f(i_j)$ at node $n(i_j)$ and arriving at time $t_f(i_k)$ at node $n(i_k)$.

Consider the example of Fig. 7. There will exist an edge between i_1 and i_3, but not between i_2 and i_3 because there is no trajectory between i_2 and i_3 obeying the second requirement. Such trajectories are not admitted in the interval graph, because they do not extend the possibilities of reaching the global goal node; if the trajectory from i_1 to i_3 is extended with two waiting steps it is equivalent with the trajectory from i_2 to i_3, in other words the first trajectory *subsumes* the latter. In fact, a vertex in the interval graph may only have more than one incoming edge when one or more trajectories with different sources are able to reach the interval at the same time t_f. Since these trajectories have the same possibilities, only one of them is kept as edge in graph. This means that the interval graph is actually a tree, which is rooted at the interval on the global start node s at start time t_0.

In our algorithm, we do not explicitly compute the interval tree, but we lazy evaluate the branches (edges) during the search for a global trajectory. For the evaluation of the different branches in the tree, we use the concept of *probes*. Each probe evaluates a branch in the tree and when a branching point (vertex) has been reached it sends out new probes on the subbranches. So during the search for a trajectory a collection of probes is to be maintained. In principle, each probe is executing Alg. 2, but it is only allowed to proceed one step at a time. The order in which the probes are proceeded is globally coordinated. For this purpose, we use an A*-like search [9], in which the probe that is most promising to find a trajectory to the global goal node is allowed to evaluate and proceed one step. This repeats until the goal node has been reached.

5.2 A probe

The probe is the main conceptual object of our algorithm. The probes explore the reachable part of the roadmap in a search for a global trajectory from the start to the goal node. Each probe is bounded to one arc of the roadmap. It is initialized with a source state-time (x_s, t_s) on that arc and it is aiming to reach the destination node in an unvisited interval. In section 4 we saw how such trajectories are computed. Each probe carries its own stack of state-times.

More than one probe may appear on the same arc, but for now we assume that the probes do not influence each other's behavior. This means that each probe has to maintain for itself

which state-times it has visited. In section 6.3 we lift this deficiency by introducing probe interaction.

Since we use an A*-search, we have to define a function $f(p)$ that determines for each probe p how promising it is. $f(p)$ gives an estimate of the cost of the time-optimal global trajectory to which p is contributing. It is computed as follows:

$$f(p) = g(p) + h(p)$$

where $g(p)$ is the cost of the trajectory between s and the current state-time of p, and $h(p)$ a *lower bound* estimate of the cost of the time-optimal trajectory between the current state-time of p and the goal node g.

Let (x_u, t_u) be the top element of p's stack. Then the value of $g(p)$ trivially evaluates to:

$$g(p) = t_u$$

The value of $h(p)$ is computed using information available in the roadmap. To estimate the amount of time a trajectory from (x_u, t_u) to g takes, we use the *roadmap distance $D(x_u, g)$* from the current state x_u of the probe to the goal node g. This roadmap distance is available if prior to the query phase a single-source Dijkstra's shortest path algorithm is carried out on the roadmap, with node g as its source. Now, a lower bound estimate for the cost of a time-optimal trajectory from (x_u, t_u) to g is:

$$h(p) = \frac{D(x_u, g)}{v_{\max}}$$

It is the amount of time needed to reach the goal node if no moving obstacles would stand in the way.

The probe with the highest *priority* is the probe with the lowest value for $f(p)$. It is the most promising node and is proceeded a step in the search-algorithm. If two probes have the same $f(p)$-value, the one with the smallest $g(p)$ is given priority.

5.3   A search over the interval tree

Consider a roadmap with start node s and goal node g and a start time t_0. The root of the interval tree is the interval on s at time t_0. From this interval, there may be edges in the interval tree to intervals on neighboring nodes, so on each outgoing arc from s a probe is released with start state-time (s, t_0) trying to reach the destination node in the first reachable unvisited interval. It will also search for next intervals, because it continues its search after it arrived at the first reachable interval on its destination node (see section 4). Also the returning trajectories must be considered, so for this purpose probes have to be launched too. These returning probes are launched on the *incoming* arcs of s, since s is their destination. Their start state-time is also (s, t_0).

All the probes are stored in a priority queue. In each step of the algorithm, the top element of the priority queue, i.e. the most promising probe with the highest priority, is allowed to proceed one step. This is in principle repeated infinitely. A number of events can occur during the algorithm:

- A probe's stack becomes empty. In this case the probe is deleted and removed from the priority queue. If it was the last probe in the queue, the algorithm terminates and reports that no trajectory exists.

- A probe reaches the global goal node. In this case the algorithm is terminated and the near-time-optimal trajectory is read out by following the backpointers.

- A probe p reaches the destination node n of its arc at time t_p in an unvisited interval. In terms of the interval tree, this means that an edge has been established to a new branching point, so new probes have to be sent out on the incident arcs of n. Advancing probes are sent out on the outgoing arcs of n with source state-time (n, t_p) and returning probes are launched on the incoming arcs with the same source state-time. The probe p itself is *not* deleted; it continues its search for next unvisited free intervals on n.

The algorithm terminates when the goal node has been found by one of the probes, or when all probes have been deleted. In the latter case there is no trajectory in the roadmap toward the goal node. However, it is also possible that no trajectory exists, but that the algorithm is running forever, with probes waiting vainly for the dynamic obstacles to step aside. Therefore, some upper bound $t_{\max}$ on the time may be set, to make sure that it terminates. If for the most promising probe holds that $f(p) > t_{\max}$, the algorithm stops and reports failure. The pseudocode of the algorithm is given in Alg. 3.

Algorithm 3 FIND GLOBAL TRAJECTORY(s, g, t_0)

1: Initialize probes on all the outgoing (advancing probes) and incoming (returning probes) arcs of s with source state-time (s, t_0) and store them in the priority queue.
2: **while** the priority queue is not empty **do**
3:   $p \leftarrow$ top element of the priority queue.
4:   **if** $f(p) > t_{\max}$ **then**
5:     Terminate algorithm. Report failure.
6:   $(x, t) \leftarrow p.\text{DoStep}()$
7:   **if** $x =$ destination node n of probe p and t is in an unvisited interval of n **then**
8:     **if** $n = g$ **then**
9:       Terminate algorithm. The trajectory is read out by following the backpointers.
10:    **else**
11:      Initialize probes on all the outgoing (advancing probes) and incoming (returning probes) arcs of n with source state-time (n, t) and append them to the priority queue.
12:   **if** $p.\text{StackEmpty}()$ **then**
13:     Delete p and remove it from the priority queue
14: Report that no trajectory exists.

It is easy to prove that this algorithm yields a near-time-optimal trajectory from the start to the goal node. In section 4 we already saw that each local trajectory from interval to interval is near-time-optimal. Since every reachable interval is considered in the algorithm, this also holds for the first reachable interval on the goal node. Hence, the trajectory found to this interval is near-time-optimal too.

5.4 Determining whether an interval is unvisited

When a probe reaches a free interval, we have to determine whether it is unvisited. For this purpose we maintain for each node in the roadmap at what times it has been visited by a

probe. When a probe arrives at the destination node n at time t and both time t and $t-1$ on n are unvisited, it is sure that an unvisited free interval is reached.

We can prove this as follows. Suppose probe p arrives at time t at an interval on n that has been visited before, but that times t and $t-1$ are unvisited on n. Then a probe p' must have visited the interval at a time $< t-1$. Since a probe reaching its destination node is not deleted and goes on with searching for new free intervals on the node, probe p' at time $< t-1$ on n had a higher priority than p, so p' is doing steps first. Probe p may arrive at (n,t) before p', but then at least $(n,t-1)$ has been visited by p'. This contradicts to our assumption, and hence the free interval at n is unvisited when a probe reaches it at time t and both time t and $t-1$ are unvisited.

6 Extensions and Optimizations

In this section we discuss some extensions and optimizations to the algorithm. They were not necessary for the understanding of our approach, but they can give a considerable improvement in terms of performance or applicability.

6.1 Directed roadmaps

We presented the method as to work only with undirected roadmaps, but the method is easily extended to work with directed roadmaps as well (in such roadmaps the robot can only move forward). Actually, the method only becomes easier. When the arcs of the roadmap are directed, the robot can not move backwards over an arc, so in search for a local trajectory there are only two choices for the velocity (v_p and 0) instead of three (v_p, 0 and $-v_p$). Also returning trajectories cannot exist, so no returning probes have to be sent on incoming arcs. The rest of the algorithm remains unchanged.

6.2 Launching probes

When a probe reaches the destination node of its arc, new probes are launched on all the incident arcs of the destination node. A few of them however, need not be sent. Suppose probe p on arc a reaches it destination node n at time t_p. Now, no returning probe needs to be sent on a, because p itself will search at that arc for next intervals on n (see Fig. 8a).

Let us call the 'opposite' arc of a going the other direction a'. Also no advancing probe needs to be sent on a', because at the time p was launched on a, a probe p' was launched on a' that subsumes a new probe (see Fig. 8b).

Furthermore, no advancing probes need to be sent on dead ends of the roadmap, i.e. arcs leading to nodes with only one incident arc. Arriving at such nodes does not extend the possibilities of reaching the goal node. Returning probes though need to be sent on these arcs. Only if the dead end leads to the goal node itself, an advancing probe has to be sent as well.

6.3 Probe interaction

Up to now we did not let the probes interact, but as multiple probes may appear on the same arc, there may occur situations in which two probes are exploring the same parts of the
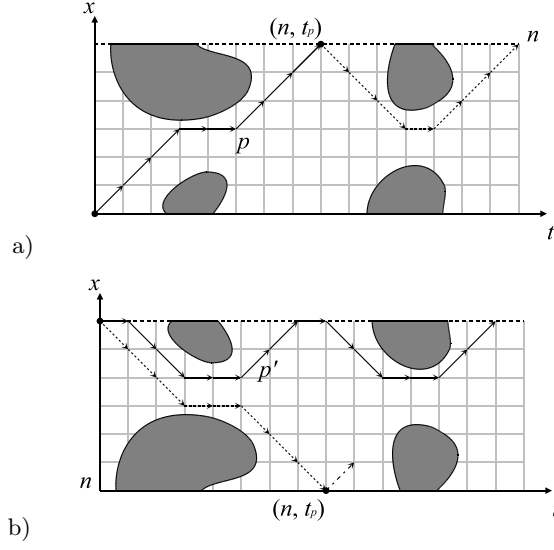
Figure 8: a) A probe p on an arc with destination node n. Since p is continuing its search after reaching n, no returning probe (with source state-time (n, t_p)) needs to be sent on this arc. b) The opposite arc with a mirrored state-time space. No advancing probe needs to be sent on this arc from (n, t_p), because it is subsumed by a returning probe p' with the same destination node, sent at the same time as p. The projection of the trajectory of p on this mirrored state-time space is indicated with a dotted line.

state-time space. This is of course unnecessary and to prevent this, one of the probes can be deleted.

Consider the example of Fig. 9. Two probes p_1 and p_2, originating from different intervals, are active on the same arc. The first state-time that is visited by both probes, is called the meeting state-time. Beyond the meeting state-time both probes will explore common parts of the state-time space. Yet, the probes are not equivalent. Probe p_2 is able to find a trajectory 'underneath' the skinny obstacle (dotted trajectory in Fig. 9), whereas probe p_1 is not, so to keep open all the possibilities probe p_2 may not be deleted.

To see why probe p_1 can be deleted, let us advance the following theorem.

Theorem 6.1. *If a probe p has visited a state-time (x, t), a trajectory for p reaching x_d before time $t + \frac{x_d - x}{v_p}$ either has already been found or does not exist.*
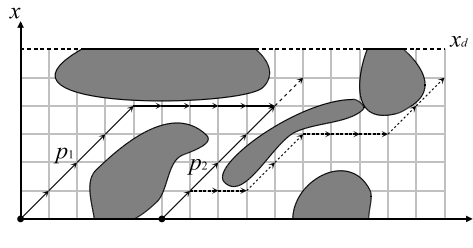

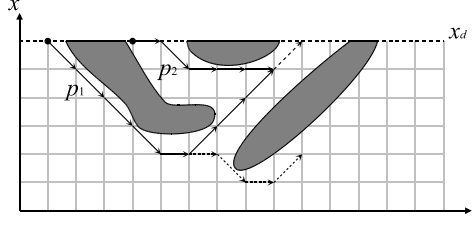Figure 9: Two probes on the same arc. Probe p_1 is deleted.

15

Figure 10: Two returning probes on the same arc. Probe p_2 is deleted.

Proof The destination node x_d is reached at time $t + \frac{x_d - x}{v_p}$ if only advancing steps were taken from (x, t). Trajectories reaching x_d before that time must already have been found if they exist. This follows from the order in which the algorithm pushes the state-times onto the stack. ∎

Now, let the meeting state-time of probes p_1 and p_2 be (x, t). According to the above theorem both probe p_1 and p_2 won't find new trajectories reaching x_d before $t + \frac{x_d - x}{v_p}$, but every trajectory of probe p_1 reaching x_d at or after that time will intersect the trace of probe p_2. So probe p_2 has at least as many possibilities as p_1. In other words, p_2 subsumes p_1 and hence in this example probe p_1 may be deleted.

In general the following rules apply to probe deletion:

- If both probes are advancing probes, the one that started earliest (see Fig. 9) can be deleted.

- If both probes are returning probes, the one that started latest can be deleted (see Fig. 10).

- If one probe is an advancing probe and one probe is a returning probe, the returning probe can be deleted.

Given the order in which the probes proceed, we can assure that the meeting state-time is always the last visited state-time of the probe that is deleted. This means that no parts of state-time space are vainly explored twice.

Before, we let each probe maintain for itself which state-times it has visited. With the given rules for probe interaction, this is not necessary anymore. It suffices to store whether the state-times have been visited and if so, by which probe.

6.4 Deleting probes

There are situations in which a probe can be deleted before its stack becomes empty. This saves a lot of unnecessary collision checks.

Consider the example of Fig. 11. In this case an advancing probe is pushed back to the source node of the arc by the state-time obstacle. It is clear that once the probe has reached this source node, there is no possibility left to reach the destination node. To prevent the probe of unnecessarily exploring the part of state-time space bounded by its own trace (light gray area in the figure), it is deleted.
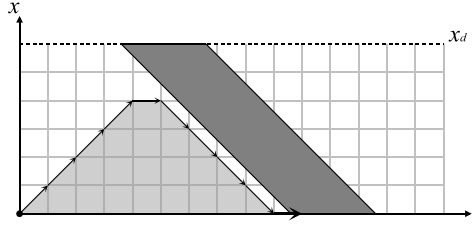
Figure 11: A situation in which a probe can be deleted before its stack becomes empty. This prevents the gray area of being explored vainly.

The exact situation in which the probe can be deleted is if it fails to do a step on the source node of the arc (thick arrow in the figure). This holds for both advancing and returning probes. Note that we refer to the source node of an arc as the other node of the arc than the destination node.

When a probe is launched on an arc, an opposite probe is launched at the same time on the opposite arc. When a probe is deleted for the reason described here, its opposite probe can be deleted as well.

6.5 Roadmap optimization

The roadmap that is used for the algorithm may be optimized in the preprocessing phase. For our algorithm, nodes with only two incident arcs need not be treated as nodes. Both incident arcs can be considered as one (hinged) arc. This saves overhead in the algorithm as well as a number of collision checks.

7 Experimental results

The algorithm has been implemented for a free-flying robot with six degrees of freedom in a three-dimensional workspace. We performed experiments in different environments and the results indicate that the method achieves interactive performance. In this section we describe one experiment in detail.

7.1 The dynamic environment

Our method was tested in the building floor scene of Fig. 12. The scene has dimensions of 8 (length) by 5 (width) by 2 (height) units of length. In the scene two dynamic obstacles A and B are moving. A moves along an H-shaped trajectory and B along a rectangular trajectory. The velocity of both dynamic obstacles is 1 unit of length per unit of time. The positions of the dynamic obstacles at the start time are shown in the figure. The motions of both obstacles are cyclic, i.e. they move infinitely.

As the robot we used a table, which has a radius of 0.5 units of length. It has to move through some narrow passages (having a width of 0.6 units of length) from s in the lower-right room to g in the left room. The distance between two configurations of the robot is measured as the euclidean distance plus the amount of rotation times the robot's radius. Its velocity under this distance measure is bounded by 1 unit of length per unit of time.
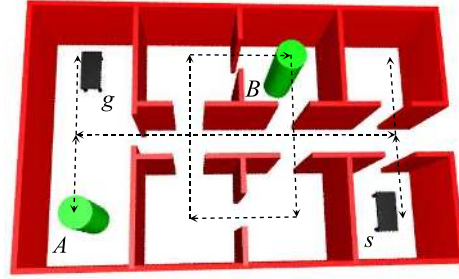
Figure 12: An environment in which a table has to move from s to g avoiding the moving obstacles A and B (cylinders). The cylinders move cyclically along the dotted lines.

Figure 13: A roadmap that is collision-free with respect to the static obstacles. The rotational degrees of freedom are not shown in the roadmap.

For the static part of the scene a roadmap was created by our motion planning system SAMPLE (System for Advanced Motion PLanning Experiments) using a variant of PRM that combines Medial Axis Sampling [14] and a node connection strategy that allows the formation of cycles in the roadmap [11]. The construction of the roadmap stopped when a predefined set of query configurations was connected by the roadmap. The roadmap is shown in Fig. 13.

The shortest path in the roadmap between the start and the goal configuration is 15.82 units of length. So if no dynamic obstacles would be present 15.82 units of time are necessary to complete the trajectory. However, dynamic obstacle A will move through the passageway in the opposite direction of the robot, so the robot must make a detour to avoid this obstacle.

7.2 Results

The running time of the algorithm is directly dependent on the choice of the value of the principal time step τ. In the experiments we chose τ to be 0.07, so that the collision checking resolution with respect to the dynamic obstacles is exactly corresponding to the resolution in which the roadmap was collision checked with respect to the static obstacles.

The algorithm was run on a 3 GHz Pentium IV with 1 GByte of memory. For the problem described above, it returned a trajectory in 0.70 seconds of computation time. The trajectory takes 35.14 units of time to traverse, and indeed the robot must make quite a detour to avoid the dynamic obstacles (see Fig. 14). It more than once 'flees' into a room at the side of the passageway. Obviously, the trajectory is collision-free with respect to both the static and the dynamic obstacles.
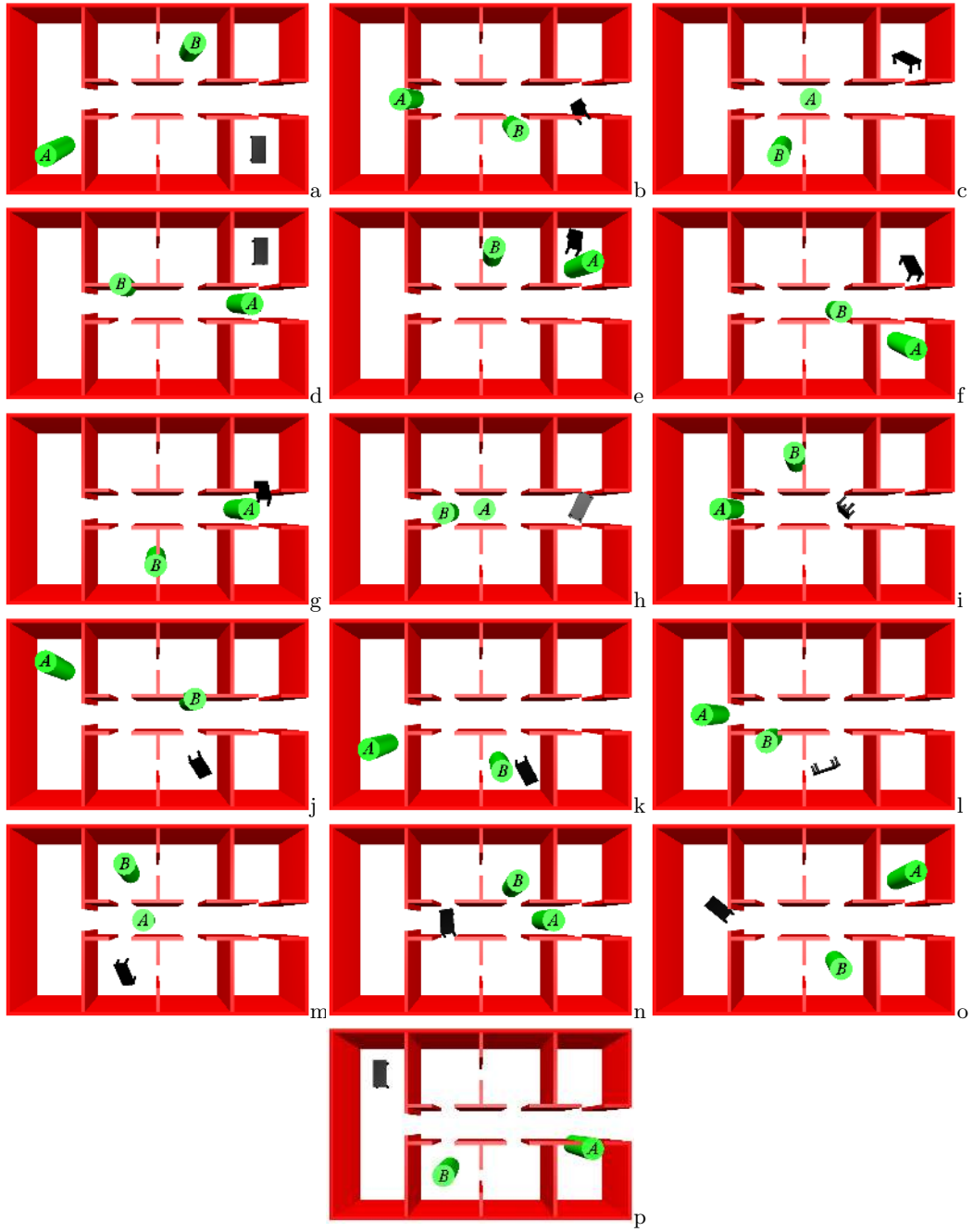
Figure 14: Pictures from the trajectory. A table has to move from the lower-right room (a) to the left room (p). The table first moves to the upper-right room to find room for avoiding dynamic obstacle A (a-e). Then it moves in the slipstream of A through the passageway (f-i). The table then moves to the lower room to find room for avoiding obstacle B (j-k). After this it moves in the slipstream of B to enter the left room (l-o). Obstacle A is moving toward the right part of the scene, so the table can safely reach its goal position (p).

We also performed an experiment for the opposite query, i.e. the roles of s and g are interchanged. In this case the robot can more or less move in the slipstream of dynamic obstacle A, and indeed the resulting trajectory is less complicated. It takes 23.87 units of time to traverse the trajectory and it was computed in 0.39 seconds. Similar performance was achieved in many other environments we experimented with.

8   Discussion and Conclusion

In this paper we presented a new method for motion planning in dynamic environments. The method finds trajectories in a given roadmap avoiding collisions with moving obstacles. It is applicable to any robot type with any number of degrees of freedom.

We used an implicit grid in state-time space to find near-time-optimal trajectories. As the principal time step τ approaches zero, the near-time-optimal trajectory becomes a time-optimal trajectory, so the parameter τ gives a trade-off between accuracy and speed. We showed in our experiments that our algorithm performs very fast even for small values of τ in complicated planning problems.

A great advantage over other methods is that the static obstacles are not of concern. Scenes often contain narrow passages through which a path is not easily found. Our method leaves this problem to a preprocessing phase, such that interactive performance can be achieved in the query phase.

The good performance of our method is explained by the A*-nature of the algorithm. Only the potentially interesting parts of the implicit interval tree are evaluated. Note that if no moving obstacles would be present, only probes along the path in the roadmap leading directly to the goal node are processed. Since the collision checks done in this case are void (there are no dynamic obstacles), the path is returned instantly. Also if all moving obstacles stay away from the optimal path, the trajectory is reported almost instantly.

An interesting open problem is the creation of smooth trajectories, for the existing methods dealing with motion planning in dynamic environments collectively fail in this matter. For the static problem a path can be smoothed using shortcuts, but for dynamic trajectories this does not work, because shortcutting is no longer a local operation. This problem could be remedied for our method by smoothing the entire roadmap or creating a smooth roadmap in the pre-processing phase, but this remains a subject of further research.

Acknowledgment

References

[1] B. Baginski; The Z^3-Method for Fast Path Planning in Dynamic Environments. *Proc. IASTED Conf. on Applications of Control and Robotics*, pp. 47-52, 1996.

[2] P. Fiorini, Z. Shiller; Time Optimal Trajectory Planning in Dynamic Environments. *Proc. IEEE Int. Conf. on Robotics and Automation*, pp. 1553-1558, 1996.

[3] Th. Fraichard; Trajectory Planning in a Dynamic Workspace: a 'State-Time' Approach. *Advanced Robotics*, 13(1):75-94, 1999.

[4] K. Fujimura; *Motion Planning in Dynamic Environments*. Springer-Verlag, Tokyo, 1991.

[5] K. Fujimura; Time-Minimum Routes in Time-Dependent Networks. *IEEE Trans. on Robotics and Automation*, 11(3):343-351, 1995.

[6] D. Hsu, R. Kindel, J.-C. Latombe, S. Rock; Randomized Kinodynamic Motion Planning with Moving Obstacles. *Int. J. Robotics Research*, 21(3):233-255, 2002.

[7] 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.

[8] R. Kindel, D. Hsu, J.-C. Latombe, S. Rock; Kinodynamic Motion Planning Amidst Moving Obstacles. *Proc. IEEE Int. Conf. on Robotics and Automation*, pp. 537-543, 2000.

[9] J.-C. Latombe; *Robot Motion Planning*. Kluwer Academic Publishers, Boston, 1991.

[10] S. M. LaValle, J. J. Kuffner, Jr.; Randomized Kinodynamic Planning. *Proc. IEEE Int. Conf. on Robotics and Automation*, pp. 473-479, 1999.

[11] D. Nieuwenhuisen, M. H. Overmars; Useful Cycles in Probabilistic Roadmap Graphs. *Proc. IEEE International Conference on Robotics and Automation*, to appear, 2004.

[12] J. Reif, M. Sharir; Motion Planning in the Presence of Moving Obstacles. *Proc. IEEE Foundations of Computer Science*, pp. 144-154, 1985.

[13] P. Švestka; *Robot Motion Planning Using Probabilistic Road Maps*. PhD thesis, Utrecht Univ., 1997.

[14] S. A. Wilmarth, N. M. Amato, P. F. Stiller; MAPRM: A Probabilistic Roadmap Planner with Sampling on the Medial Axis of the Free Space. *Proc. IEEE Int. Conf. on Robotics and Automation*, pp. 1024-1031, 1999.