Roadmap-Based Motion Planning in Dynamic Environments

Jur P. van den Berg and Mark H. Overmars

***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 approximately 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 two-level search for a shortest path. On the local level, trajectories on single edges of the roadmap are found using a depth-first search on an implicit grid in state-time space. On the global level, these local trajectories are coordinated using an A*-search to find a global trajectory to the goal configuration. 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 both free-flying and articulated robots in three-dimensional workspaces, and it has been applied to multirobot motion planning, as well. Experiments show that the method achieves interactive performance in complex environments.**

***Index Terms*—Dynamic environments, motion planning, multiple robots.**

I. 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 [10]. A popular approximate method is the probabilistic roadmap planner (PRM) [9], [15]. 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], [3]–[8], [14], 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 [11] have been the methods of choice for this type of problem [1], [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 are 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.

In this paper, we propose a new method which is based on a roadmap built in a preprocessing phase. The roadmap is built for 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, then, only the dynamic obstacles need to be 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. Previous techniques, for example, [5], 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.

We use a two-level approach to find a trajectory. On the local level, trajectories on single edges of the roadmap are found in an implicit grid in state-time space. The state-time space is discretized in a similar way as [5], but we use a more efficient *depth-first* search strategy. On the global level, the local trajectories are coordinated using an A*-like search to find a near-time-optimal global trajectory in the entire roadmap.

Our method uses principles similar to those of a theoretical method of Fujimura [7], which computes an exact time-optimal path in a roadmap. On the local level, however, his method uses visibility graphs in state-time space. This requires the state-time space to be constructed explicitly, and therefore, his method works only for point robots in two-dimensional (2-D) environments where the dynamic obstacles are constrained to piecewise linear motions without rotation. To our knowledge, the method was never implemented or used in practice.

Manuscript received December 13, 2004. This paper was recommended for publication by Associate Editor K. Lynch and Editor S. Hutchinson upon evaluation of the reviewers' comments. This work was supported by the IST Programme of the EU as a Shared-Cost RTD (FET Open) Project under Contract IST-2001-39250 (MOVIE—Motion Planning in Virtual Environments). This paper was presented in part at the IEEE International Conference on Intelligent Robots and Systems, Sendai, Japan, 2004.

The authors are with the Institute of Information and Computing Sciences, Utrecht University, 3508 TB Utrecht, The Netherlands (e-mail: berg@cs.uu.nl; markov@cs.uu.nl).

Digital Object Identifier 10.1109/TRO.2005.851378

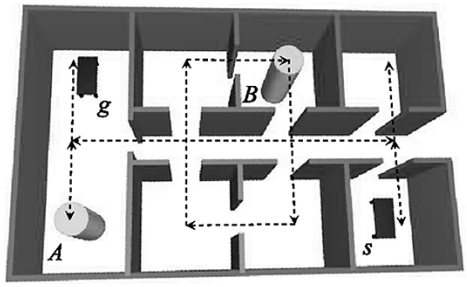

Fig. 1. Dynamic environment in which a table has to move from s to g avoiding the moving obstacles A and B (cylinders). The cylinders move cyclically along the dotted lines.

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 of whether the robot is collision-free. As in [7], we do not put constraints on the robot's motion, except for an upper bound on its velocity.

The method has been implemented for both free-flying robots and articulated robots with 6 degrees of freedom (DOFs), and various experiments show that the method achieves interactive performance in confined dynamic environments (see, e.g., Fig. 1). One of the main applications of motion planning in dynamic environments is multirobot motion planning [2]. Using our method, we were able to solve difficult planning problems involving many robots within mere seconds of computation time.

The rest of the paper is organized as follows. A formal definition of the problem is given in Section II. In Section III, we describe the global approach of our method. The problem is split up in two parts: finding local trajectories on single edges of the roadmap and finding a global trajectory through the entire roadmap. These will be discussed in Sections IV and V, respectively. In Section VI, some extensions and optimizations to the algorithm are discussed, and Section VII describes the experimental results and the application to multirobot motion planning.

II. Problem Description

A. 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 DOFs. C_{free} denotes the subset of C containing all *collision-free* configurations of the robot. In terms of C, the motion-planning problem is 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* [5]. It consists of pairs (c, t), where c 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 subject to constraints on its motion. Also, it cannot go back in time. To accentuate this difference, we define a *path* as being a continuous sequence of configurations, and a *trajectory* as a path parametrized by time such that the dynamic constraints are obeyed.

B. The Roadmap

Our method requires a roadmap that is constructed for the static part of the scene, that is, its set of vertices and edges in the configuration space C must be collision-free with respect to the*static* obstacles. This means that the roadmap can be constructed in a preprocessing phase. The start and goal configurations are assumed to be present in the roadmap as vertices. If not, they can be connected to the roadmap in the query phase. Our method is applicable to both directed and undirected roadmaps, but in this paper, we confine ourselves to the more general case of undirected roadmaps.

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. Moreover, narrow passage problems raised by the static obstacles are solved in the preprocessing phase, which substantially relieves the query phase. 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 1-D 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 [7].

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 [13]. 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 important that the roadmap contains cycles (see, e.g., [12]).

C. 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 2-D or 3-D workspace containing both static and dynamic obstacles, and let

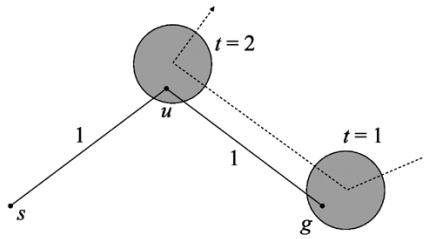

Fig. 2. Small example roadmap with two edges (solid lines), and a dynamic obstacle (gray disc) moving over the dotted trajectory. A point robot moving from s to g has to wait until the dynamic obstacle has cleared u, before advancing to g.

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 vertex 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 $cf(c,t)$ that, given a state $c \in C$ and a moment in time t, reports whether the robot configured at c collides with any moving obstacle at time t.

III. 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 vertices of the roadmap. We illustrate this using a simple example roadmap created for a point robot (see Fig. 2). It consists of three vertices s, u, and g, and edges between s and u, and u and g (solid lines in the figure). It takes one unit of time for the robot to traverse each edge. An obstacle (gray disc in the figure) is moving over g at time $t = 1$ to u at $t = 2$, and then moves away from the roadmap (dotted trajectory in the figure). If the robot starts at s at $t = 0$, it is able to reach u 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 edge (s, u) and arrive at u at $t > 2$, the path to g is free. It would, however, not be useful to arrive even later at u, because the robot would then arrive in the same *free interval* on u. A free interval on a vertex u is defined as follows.

Definition III.1: A free interval on a vertex u is a maximal continuous segment in time in which the robot configured at u is collision-free.

It is easy to see that it is not useful to arrive later in the same free interval. If the robot arrives early at the interval, it can wait on the vertex 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 vertices by modeling each free interval on a vertex as a *node* in an implicit (directed) tree, which we will call the *interval tree*. The interval tree is rooted at the interval on the start vertex s around the start time t_0, and branches exist in the tree from one node to the other when there is an edge between the corresponding vertices in the roadmap and an appropriate trajectory exists between the associated intervals.[1] A (global) trajectory between the start and goal vertex is contained in the interval tree.

Our approach uses the interval tree to find a trajectory toward the goal vertex. We do not compute the interval tree explicitly, but explore it in a lazy fashion by sending so-called *probes* through the roadmap. Probes search for trajectories between free intervals on neighboring vertices (i.e., branches in the tree). We call such trajectories *local trajectories*.

We will describe our algorithm in two stages. The first deals with the behavior of a single probe, i.e., computing a feasible local trajectory on a single edge of the roadmap using a depth-first search. In the second stage, we discuss the global probe management, in which we search the interval tree using an A*-like search to find a *global* trajectory to the goal vertex.

In the example of Fig. 2, only normal local trajectories were considered, i.e., trajectories that originate at one vertex of an edge and advance to the other vertex of the edge. 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 vertex they originate from. They first move away from the vertex along an edge to make room for a moving obstacle, after which they return to the vertex.

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

IV. Local Trajectories

In this section, we discuss how an approximately time-optimal local trajectory is computed along a single edge of the roadmap. We follow an approach similar to [5] by discretizing the state-time space into a grid. The only dynamic constraint the robot is subject to is a bound $v_{\max}$ on its maximum velocity.

We assume that the roadmap edges are undirected. For conceptual clarity though, they are considered as two separate "directed" edges in the rest of this paper, such that each edge has its own *source* and *destination vertex*.

A. The State-Time Grid

Since we consider trajectories along an edge of the roadmap, a configuration of the robot is reduced to a single variable representing the distance traveled along the edge. We denote this variable by x. It ranges from 0 to 1, where $x = 0$ and $x = 1$, respectively, correspond to the configurations on the source and destination vertex of the edge. The resulting state-time space is 2-D and consists of pairs $(x,t) \in [0,1] \times [t_0, \infty)$. Obstacles in the 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 Δt 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 $k\Delta t$, where k is an integer. Let l be the length of the edge. Given a state-time (x,t),

[1]Throughout this paper, we will use the terms vertices and edges when we refer to the roadmap, and nodes and branches when we refer to the interval tree.

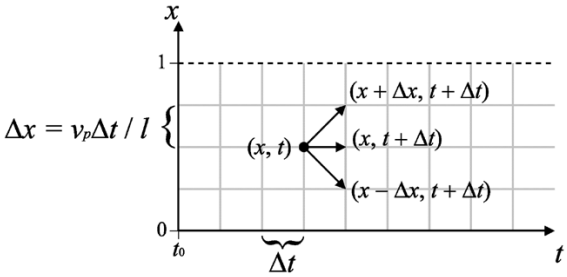

Fig. 3. State-time grid of a roadmap edge. It shows which neighbors are reachable from a given state-time (x, t).

new state-times of the robot are calculated using the following equation of motion:

$$v \in \{v_p, 0, -v_p\}$$
$$x(t + \Delta t) = x(t) + v\Delta t/l.$$

This results in a regular 2-D grid of state-times (i.e., points in state-time space) in which the robot can be. The spacings in the grid are Δt along the time axis and $\Delta x = v_p \Delta t/l$ along the state axis (see Fig. 3). We divide $v_p \Delta t$ by l here to transform the actual covered distance to the appropriate step Δx on the range [0, 1]. We choose v_p to have the largest value smaller than $v_{\max}$, such that $1/\Delta x$ is an integer. This means that the destination vertex of the edge can be reached exactly in an integer number of steps.

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 $(x + \Delta x, t + \Delta t), (x, t + \Delta t)$, and $(x - \Delta x, t + \Delta t)$ (see Fig. 3). This defines a connectivity on the state-time grid, in which the local trajectories can be found. Note that the grid is not explicitly constructed.

B. Finding a Local Trajectory

The problem of finding a local trajectory is defined as follows. Given an edge and an initial state-time (x_s, t_s), find a path in the grid to the first reachable free interval on the destination vertex. The destination vertex is reached when $x = 1$. In case of an advancing trajectory, $x_s = 0$, and in case of a returning trajectory $x_s = 1$. To prevent the algorithm from immediately returning success in the latter case, we state that the destination vertex must be reached in an *unvisited* free interval. (How to determine whether an interval has been visited is discussed in Section V-D, as it relates to the global algorithm.)

An approximately time-optimal local trajectory can be found by finding a shortest path from (x_s, t_s) to $x = 1$ in the state-time grid. In [5], an A*-algorithm is used to find the shortest path, but in our case, it can be implemented more efficiently using a *depth-first search*; this requires fewer collision checks (as we will see shortly), and the elementary operations on the datastructure (a stack instead of a priority queue) are cheaper.

The algorithm is initialized with the 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 $(x - \Delta x, t + \Delta t), (x, t + \Delta t)$ and $(x + \Delta x, t + \Delta t)$ are pushed onto the stack *in this particular order* (see Algorithm 1). This means that the most promising step (advancing toward the destination vertex), which is pushed last on the stack, is considered first. The algorithm runs until the stack is empty or the state $x = 1$ has been reached in an unvisited free interval on the destination vertex (see Algorithm 2). Backpointers and information about whether a state-time has been visited before, are maintained.

Algorithm 1 DoStep()

```
(x, t) ← StackPop()
if not (x, t).visited and cf(x, t) then
    StackPush(x − Δx, t + Δt)
    (x − Δx, t + Δt).backpointer ← (x, t)
    StackPush(x, t + Δt)
    (x, t + Δt).backpointer ← (x, t)
    StackPush(x + Δx, t + Δt)
    (x + Δx, t + Δt).backpointer ← (x, t)
    (x, t).visited ← true
    return (x, t)
else
    (x, t).visited ← true
    return NULL
```

Algorithm 2 FindLocalTrajectory(x_s, t_s)

```
StackPush(x_s, t_s)
repeat
    (x, t) ← DoStep()
until (x = 1 and the interval on the destination vertex at
  time t is unvisited) or StackEmpty()
```

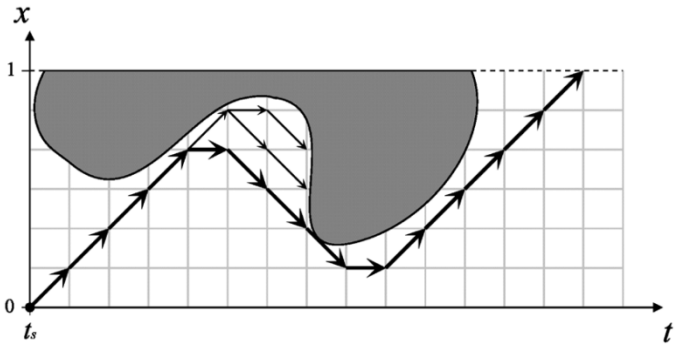

Fig. 4. Finding the shortest trajectory from $(0, t_s)$ to $x = 1$ using depth-first search. Thin arrows indicate the space explored by the algorithm, and thick arrows form the resulting trajectory. The gray object is a state-time obstacle.

We assume that the time step Δt 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 edges of the roadmap are checked for collisions [15].

Fig. 4 shows the working of the depth-first algorithm in an example state-time grid. The thin arrows indicate the space explored by the algorithm, and the thick arrows form the resulting trajectory. It is guaranteed that the depth-first search method will not explore parts of the state-time space "beneath" the resulting trajectory. Let us look at what an A*-algorithm would do in the same example (see Fig. 5). The A* method needs a lower bound estimate of the cost (i.e., the amount of time) of going from a state-time (x, t) to the destination vertex. We used $((1 - x)/\Delta x)\Delta t$, which is the optimal estimate. Yet, the A* method explores a considerably larger part of the state-time space, and hence, performs many more collision checks, to eventually find the same trajectory as the depth-first search method.

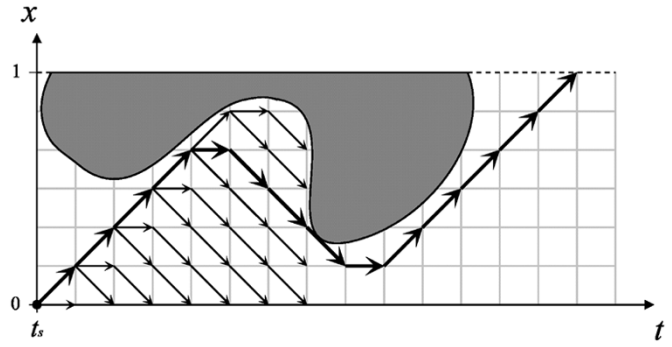

Fig. 5. Finding a trajectory in the same state-time grid as Fig. 4 using an A*-search.

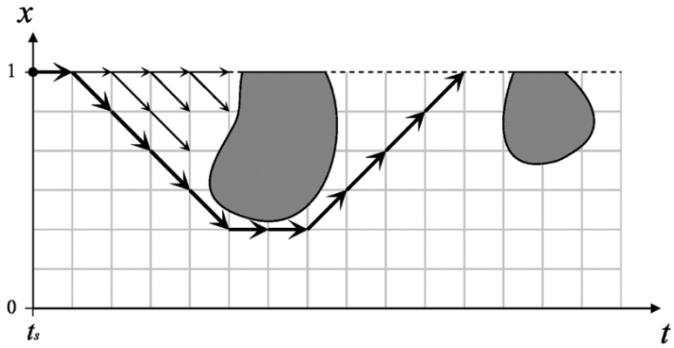

Fig. 6. Finding a returning trajectory originating at $(1, t_s)$ and arriving at $x = 1$ in an unvisited free interval of the destination vertex.

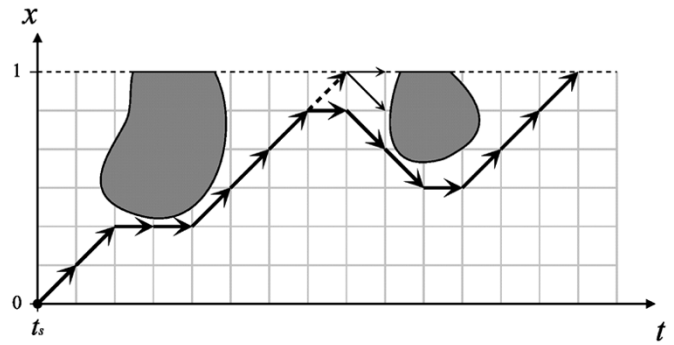

Fig. 7. Finding a trajectory to the second reachable interval on the destination vertex. The dotted arrow indicates the trajectory to the first reachable interval.

We can prove that the depth-first algorithm is correct, i.e., it yields a trajectory arriving as early as possible on the destination vertex. Let t_f be the time at which the destination vertex is first reachable from a given initial state-time (x_s, t_s). Hence, there exists at least one trajectory between (x_s, t_s) and $(1, t_f)$ in the grid. Let $\mathcal{T}$ be the set of all trajectories between (x_s, t_s) and $(1, t_f)$. 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 $T(t) \in [0, 1]$ 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 IV.1: Given an initial state-time (x_s, t_s) and an edge of which the destination vertex is earliest reachable at time t_f, the above algorithm finds the highest trajectory T_h between (x_s, t_s) and $(1, t_f)$.

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 initial state-time (x_s, t_s) also lies on T_h, the algorithm will then find a trajectory from (x_s, t_s) to $(1, t_f)$ 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 $(x+\Delta x, t+\Delta t)$. In this case, the algorithm will follow T_h, since it is the most promising step.
- The successor on T_h is $(x, t + \Delta t)$. The algorithm at this point first proceeds to state-time $(x + \Delta x, t + \Delta t)$. Suppose the algorithm finds a trajectory to $x = 1$ from this state-time. Then, this trajectory can not reach $x = 1$ before t_f, because t_f is the first time at which $x = 1$ 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 $(1, t_f)$. Hence, no trajectory is found at all from $(x+\Delta x, t+\Delta t)$. So, eventually, the algorithm returns to state-time (x, t) and now evaluates $(x, t+\Delta t)$, which is also the successor of (x, t) on T_h.
- The successor on T_h is $(x - \Delta x, t + \Delta t)$. The algorithm now first proceeds to $(x+\Delta x, t+\Delta t)$, and then to $(x, t+\Delta t)$, but for the same reason as above, it will not find a trajectory in either of these cases. So, eventually, the algorithm evaluates $(x - \Delta x, t + \Delta t)$, 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. 6). 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 V-D.

The algorithm described above finds a trajectory to the first reachable free interval on the destination vertex. However, in the example of Section III, we saw that the destination vertex 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 Algorithm 2), a trajectory to the next intervals will be found as well (see Fig. 7).

As proved above, Algorithm 2 finds the shortest path in the grid. The associated trajectory is *approximately* optimal when abstracting from the grid, but as the time step Δt approaches zero, the trajectory approaches the continuous time-optimal trajectory. For smaller Δt the algorithm obviously becomes slower, so the choice of Δt gives a tradeoff between accuracy and speed.

V. Global Trajectories

In the previous section, we discussed how to compute a local trajectory on a single edge. In this section, we will show how

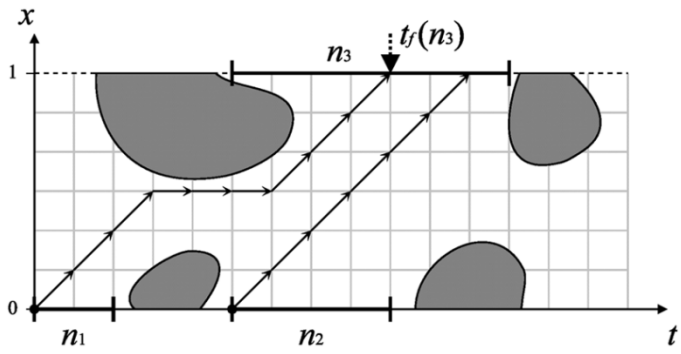

Fig. 8. State-time grid showing free intervals on the vertices and trajectories between them. A branch from n_2 to n_3 is not admitted in the interval tree.

the global trajectory from a start vertex s to a goal vertex g through the roadmap is found. As with the local trajectories, the algorithm will find an approximately time-optimal trajectory.

A. The Interval Tree

In Section III, we already gave a short introduction to the implicit interval tree. Each node in the interval tree corresponds to a free interval on a vertex in the roadmap. The root of the tree is the interval on the global start vertex s at start time t_0. An important notion is the time at which the interval associated with a node n is *first reachable* for the robot (see Fig. 8). We denote this time by $t_f(n)$. The vertex associated with node n is denoted $u(n)$.

A branch exists in the interval tree from a node n_i to a node n_j when all of the following conditions hold.

- Their associated vertices $u(n_i)$ and $u(n_j)$ are either connected by an edge in the roadmap, or $u(n_i) = u(n_j)$.
- There exists a local trajectory starting at time $t_f(n_i)$ on vertex $u(n_i)$ and arriving at time $t_f(n_j)$ on vertex $u(n_j)$.
- No other trajectory has already been found that arrives at time $t_f(n_j)$ on vertex $u(n_j)$.

Consider the example of Fig. 8. There will exist a branch from n_1 to n_3, but not from n_2 to n_3, because there is no trajectory between the intervals of n_2 and n_3 obeying the second requirement. Such trajectories need not be incorporated in the interval tree, because they do not extend the possibilities of reaching the global goal vertex; if the trajectory between the intervals of n_1 and n_3 is extended with two waiting steps, it is equivalent with the trajectory between the intervals of n_2 and n_3. In other words, the first trajectory *subsumes* the latter.

The third condition is posed as multiple trajectories with different sources may reach the interval at the same time t_f. Since these trajectories have the same possibilities, only one of them needs to be kept as branch in the tree.

In our algorithm, we do not explicitly compute the interval tree, but we lazily evaluate the branches during the search for a global trajectory. To this end, we use the concept of *probes*. Each probe evaluates a branch in the tree (i.e., it tries to find a local trajectory meeting the above three criteria), and when a node 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 Algorithm 2, but it is only allowed to proceed one step at a time. The order in which the probes proceed is globally coordinated. For this purpose, we use an A*-like search [10], in which the probe that is most promising to find a trajectory to the global goal vertex is allowed to evaluate and proceed one step. This repeats until the goal vertex has been reached.

B. 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 vertex. Each probe p is bounded to one edge, say (u_s, u_d), of the roadmap. A probe is initialized with an initial state-time (x_s, t_s) on (u_s, u_d), and is aiming to reach the destination vertex u_d in an unvisited interval. In Section IV, 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 edge, but for now, we assume that the probes do not influence each other's behavior. In Section VI-B we discuss how multiple probes are coordinated on an edge.

Since we use an A*-search to coordinate the order in which the probes proceed, 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 the start vertex s and the current state-time of p, and $h(p)$ is a *lower-bound* estimate of the cost of the time-optimal trajectory between the current state-time of p and the goal vertex g.

Let (x, t) be the top element of p's stack. Then the value of $g(p)$ trivially evaluates to

$$g(p) = t.$$

The value of $h(p)$ is computed as the sum of the estimated amount of time it takes for the probe to reach its destination vertex u_d, and the estimated amount of time it takes to go from u_d to the goal vertex g. For the latter term, we use the *roadmap distance* $D(u_d, g)$ between u_d and 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 vertex g as its source. The total lower-bound estimate for the cost of a time-optimal trajectory from (x, t) to g is

$$h(p) = \frac{1 - x}{\Delta x}\Delta t + \frac{D(u_d, g)}{v_{\max}}.$$

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

The probe with the highest *priority*, i.e., the most promising probe, is the probe with the lowest value for $f(p)$. If two probes have the same $f(p)$-value, the one with the smallest $h(p)$ is given priority.

C. Finding a Global Trajectory

Consider a roadmap with start vertex s and goal vertex 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 branches in the interval tree to intervals on neighboring vertices, so on each outgoing edge (s, u) from s a probe is released with initial state-time

$(0, t_0)$ trying to reach unvisited free intervals on the edge's destination vertex u. It is important to note that after a probe arrives at the first reachable interval on its destination vertex, it continues its search for the next intervals (see Fig. 7). 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* edges (u, s) of s, since s is their destination. Their initial state-time is $(1, t_0)$.

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 vertex. 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 vertex u_d of its edge (u_s, u_d) at time t_p in an unvisited interval. In terms of the interval tree, this means that a branch has been established to a new node, so new probes have to be sent out on the incident edges of u_d. Advancing probes are sent out on the outgoing edges (u_d, u) with initial state-time $(0, t_p)$, and returning probes are launched on the incoming edges (u, u_d) with initial state-time $(1, t_p)$. The probe p itself is *not* deleted; it continues its search for next unvisited free intervals on u_d.

The algorithm terminates when the goal vertex has been found by one of the probes, or when all probes have been deleted. In the latter case, there is no trajectory toward the goal vertex. 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, it holds that $f(p) > t_{\max}$, the algorithm stops and reports failure. The pseudocode of the algorithm is given in Algorithm 3.

It is easy to prove that this algorithm yields an approximately time-optimal trajectory from the start to the goal vertex. In Section IV, 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 vertex. Hence, the trajectory found to this interval is near-time-optimal too.

Algorithm 3 FINDGLOBALTRAJECTORY(s, g, t_0)

```
1: Initialize advancing probes on all the outgoing edges
   (s,u) of s with state-time (0,t_0) and returning probes
   on incoming edges (u,s) of s with state-time (1,t_0), and
   store them in the priority queue.
2: while the priority queue is not empty do
3:    p ← top element of the priority queue.
4:    (u_s,u_d) ← edge on which p is active.
5:    if f(p) > t_max then
6:       Terminate algorithm. Report failure.
7:    (x,t) ← p.DoStep()
8:    if x = 1 and u_d = g then
9:       Success! Terminate algorithm. The trajectory is read
         out by following the backpointers.
10:   if x = 1 and t is in an unvisited interval on u_d then
11:      Initialize advancing probes on all the outgoing edges
         (u_d,u) of u_d with state-time (0,t) and returning
         probes on incoming edges (u,u_d) of u_d with state-
         time (1,t), and append them to the priority queue.
12:   if p.StackEmpty() then
13:      Delete p and remove it from the priority queue
14: Report that no trajectory exists.
```

D. 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 vertex in the roadmap what times it has been visited by a probe. When a probe arrives at the destination vertex u_d at time t, and both time t and $t - \Delta t$ on u_d 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 u_d that has been visited before, but that times t and $t - \Delta t$ are unvisited on u_d. Then a probe p' must have visited the interval at a time $< t - \Delta t$. Since a probe reaching its destination vertex is not deleted and goes on with searching for new free intervals on the vertex, probe p' at time $< t - \Delta t$ on u_d had a higher priority than p, so p' is doing steps first. Probe p may arrive at time t on u_d before p' does, but then at least time $t - \Delta t$ on u_d has been visited by p'. This contradicts our assumption, and hence, the free interval on u_d is unvisited when a probe reaches it at time t and both time t and $t - \Delta t$ are unvisited.

VI. 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.

A. Launching Probes

In the current algorithm, when a probe reaches the destination vertex of its edge, new probes are launched on all the incident edges of the destination vertex. A few of them, however, need not be sent. Suppose probe p on edge (u_s, u_d) reaches it destination vertex u_d at time t_p. Now, no returning probe needs to be sent on (u_s, u_d), because p itself will search at that edge for next intervals on u_d [see Fig. 9(a)].

Also, no advancing probe needs to be sent on the "opposite" edge (u_d, u_s), because at the time (an advancing probe) p was launched on (u_s, u_d), a (returning) probe p' was launched on (u_d, u_s) trying to reach u_s as well. This probe p' subsumes a new probe [see Fig. 9(b)].

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

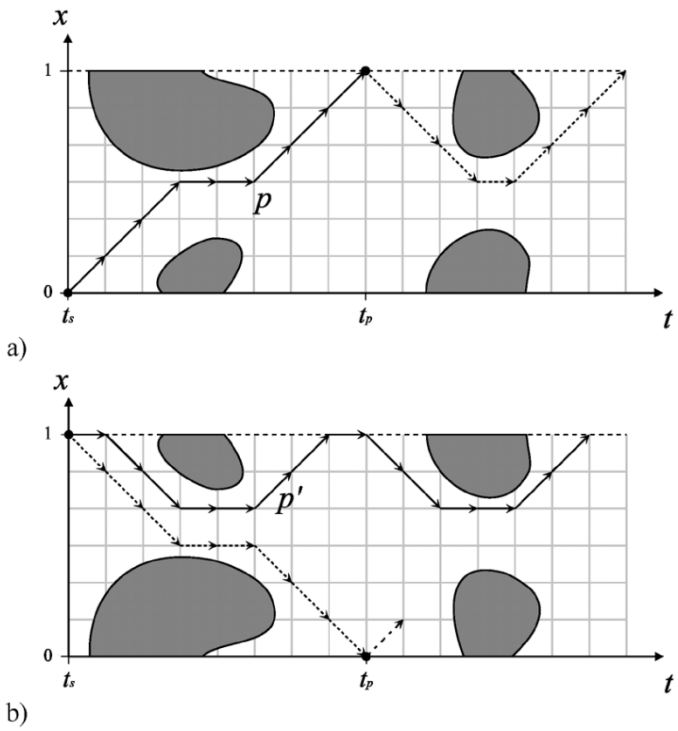

Fig. 9. (a) Probe p active on an edge. Since p is continuing its search after reaching its destination vertex, no returning probe (with initial state-time $(1, t_p)$) needs to be sent on this edge. (b) The opposite edge with a mirrored state-time space. No advancing probe needs to be sent on this edge from $(0, t_p)$, because it is subsumed by a returning probe p', which was 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.

B. Probe Coordination on an Edge

Up to now, we did not let the probes influence each other's behavior, but as multiple probes may appear on the same edge, they may explore common parts of the state-time space. To prevent this, we allow only one of the probes to proceed, in case multiple probes are active on the same edge. To be precise, on each edge, at most one probe is represented in the global priority queue that coordinates the order in which the probes proceed.

Consider the example of Fig. 10. Two returning probes p_1 and p_2, and two advancing probes p_3 and p_4, originating from different intervals, are active on the same edge (u_s, u_d). All of these probes aim to reach the same destination vertex u_d in the first reachable free interval. Each of these probes has a different *start time*, i.e., the time at which the probe was launched. We now sort the probes in a list according to the following rules.

- Returning probes precede advancing probes.
- Returning probes are sorted in *reverse order* of their start times, i.e., the returning probe that started latest is in front of the list.
- Advancing probes are sorted in order of their start times, i.e., the advancing probe that started latest is at the end of the list.

So, for the edge in our example, the list would be $\{p_2, p_1, p_3, p_4\}$. We now only allow the probe in front of the list to do steps on the edge, i.e., only this probe is present in the global priority queue as a representative of this edge. The rationale behind this is the following. We can prove that the probe in front of the list will either find a trajectory to the first reachable free interval on the destination vertex, or its stack becomes empty (see Fig. 10). In the latter case, the probe is deleted and the second probe in the list becomes the top probe.

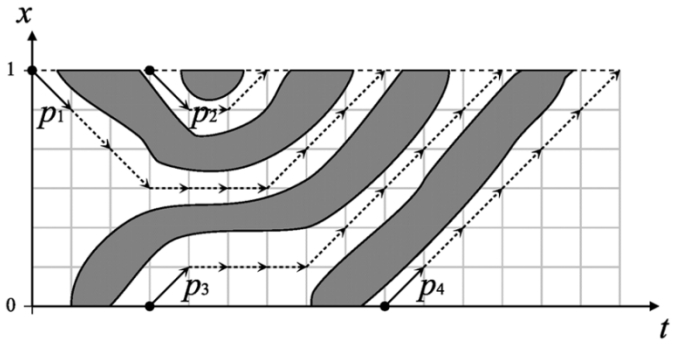

Fig. 10. Four probes on the same edge, and their projected traces (dotted).

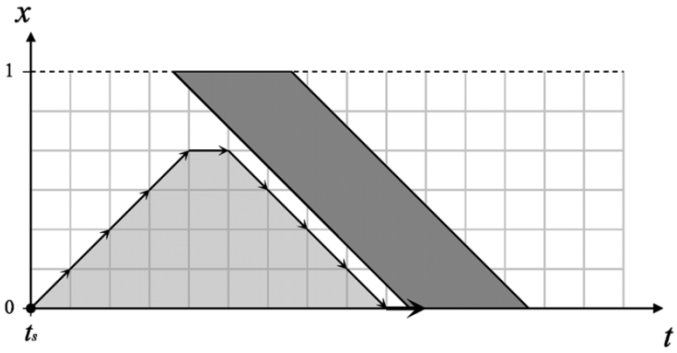

Fig. 11. Situation in which a probe can be deleted before its stack becomes empty. This prevents the gray area from being explored vainly.

None of the probes are allowed to explore parts of the state-time space that were visited before by other probes.

C. 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 vertex u_s of the edge (u_s, u_d) by the state-time obstacle. It is clear that once the probe has reached this source vertex, there is no possibility left to reach the destination vertex. Yet, the probe's stack still contains state-times. 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.

The exact situation in which the probe can be deleted is if it fails to do a step on the source vertex of the edge (thick arrow in the figure). This holds for both advancing and returning probes. In fact, when a probe is deleted for this reason, all probes that are active on the same edge can be deleted, because, given the order in which they are sorted in the list, they would need to cross the trace of the deleted probe to reach the destination vertex.

VII. Experimental Results

The algorithm has been implemented for both free-flying and articulated robots with six DOFs in a 3-D workspace. We performed experiments in different environments, and the results

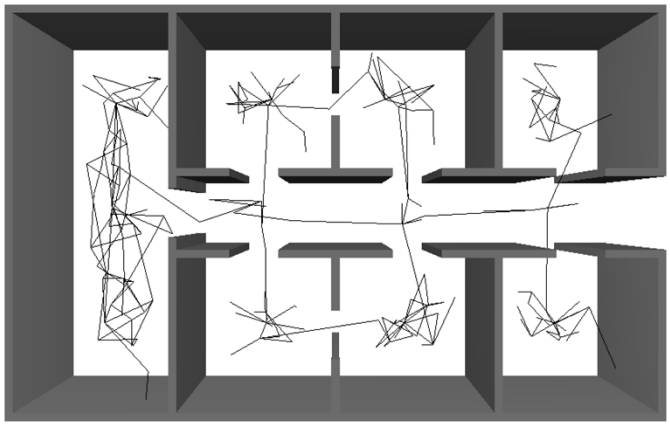

Fig. 12. Roadmap that is collision-free with respect to the static obstacles. The rotational DOFs are not shown in the roadmap.

indicate that the method achieves interactive performance. In this section, we describe a number of experiments in detail, and discuss the application of our method to multirobot motion planning as well.

A. A Preliminary Experiment

Our method was tested in the building floor scene of Fig. 1. 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 free-flying 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.

For the static part of the scene, a roadmap was created using a variant of PRM that allows the formation of cycles in the roadmap [12]. The construction of the roadmap stopped when a predefined set of query configurations was connected by the roadmap. The roadmap is shown in Fig. 12 and consists of 198 vertices and 478 edges.

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.

This example problem may look simple, but it certainly is not. The static obstacles in the scene strongly confine the maneuverability of the robot, and the environment contains many narrow passages. Moreover, the dynamic obstacles "sweep clean" the passageways in the environment, so the robot really has to "flee" into other rooms to avoid collisions.

The running time of the algorithm is directly dependent on the choice of the value of the principal time step Δt. In this experiment, we chose Δt 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 GB of memory. For the problem described above, it returned a trajectory in only 0.46 s 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. 13). 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.

The computed trajectory is near-time-optimal over the roadmap. Yet, it takes 35.14 units of time to traverse, while this would be 15.82 units of time without obstacles. We call these numbers the *trajectory length* and the *roadmap distance*, respectively, and the ratio between them the *delay factor*. It gives an indication of the difficulty of the problem. For this experiment, the delay factor is $35.14/15.82 = 2.22$.

B. Varying the Quantities

In this section, we explore the effect on the performance of our method of gradually increasing major quantities, among which the delay factor of the problem, the size of the roadmap, and the value of the time step Δt. For these experiments, we use the environment and the quantities of the above experiment. Only the quantity under consideration is varied.

Difficulty: The difficulty of a problem is always hard to measure, let alone varying it *ceteris paribus*, i.e., without changing other quantities. We use the delay factor as a measure for the difficulty of a problem. In the above experiment, the delay factor was 2.22.

We vary the delay factor over the experiments by tuning the velocity and the behavior of the dynamic obstacles. The other quantities are kept equal; we use the same roadmap over the experiments and do not change the value of Δt.

The performance of our method not only depends on the delay factor, but also on whether an obstacle interferes with the robot in the beginning of the trajectory, when few probes are active, or near the goal configuration, when many probes are active. To cover a large range of possibilities, we performed 55 experiments with different behavior of the dynamic obstacles.

One would expect that the running time grows exponentially with the delay factor, because the more time it takes to reach the goal, the wider the interval tree grows. The A*-nature of our algorithm, though, moderates this effect as it focuses the search toward the goal. Also, for larger delay factors, the space gets more confined. This means that there is less maneuverability for the robot, which has a moderating effect on the width of the interval tree (and hence, the number of probes) as well.

Fig. 14 gives the result of the experiments. The scatter-plot indicates that the running time actually grows more or less linear with the delay factor. It turned out to be quite hard to find problems with a delay factor larger than six for which a solution still

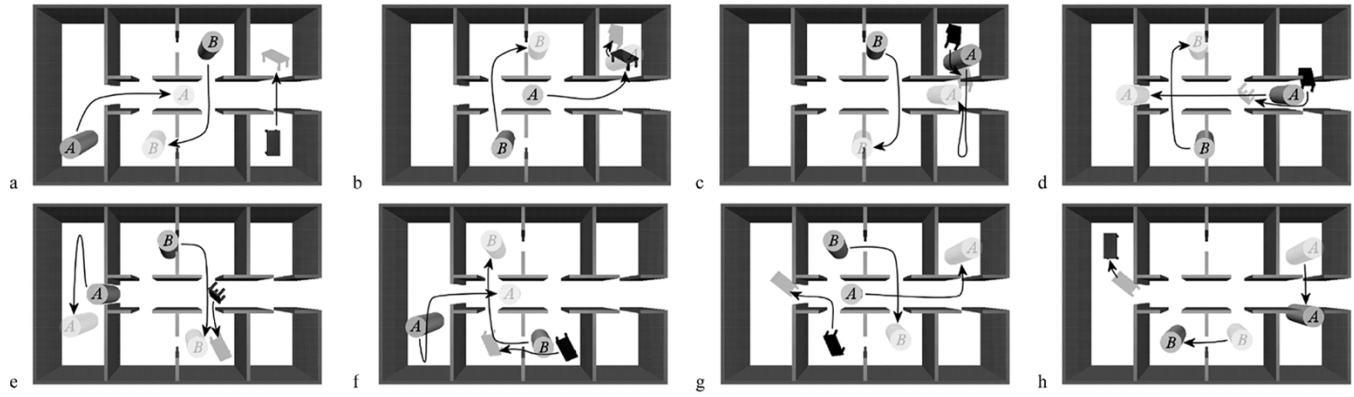

Fig. 13. Pictures from the trajectory. A table has to move from the lower-right room (a) to the left room (h). The table first moves to the upper-right room to find room for avoiding dynamic obstacle A (a-b). Then it moves in the slipstream of A through the passageway (c–d). The table then moves to the lower room to find room for avoiding obstacle B (e). After this, it moves in the slipstream of B to enter the left room (f). Obstacle A is moving toward the right part of the scene, so the table can safely reach its goal position (g–h).

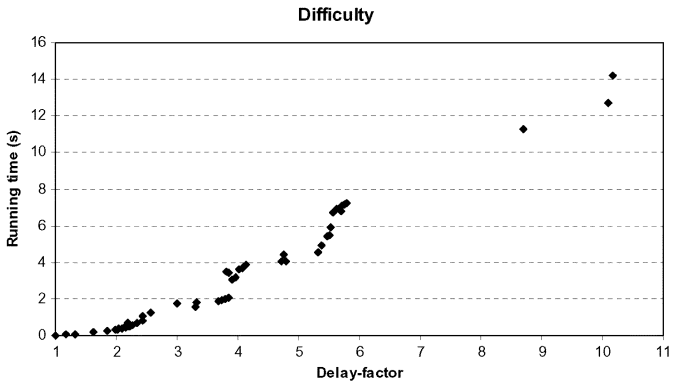

Fig. 14. Scatter-plot of 55 experiments with different delay factors.

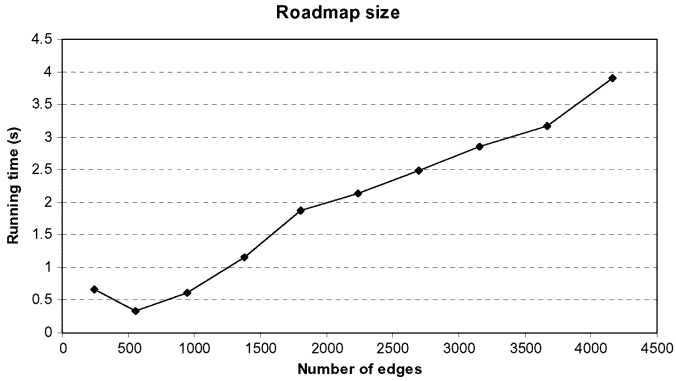

Fig. 15. Running time for roadmaps of various sizes.

exists. The few problems we did find, however, confirm the apparent linear relationship. Note that when the delay factor is 1, the running time is nearly zero.

Roadmap Size: The size of the roadmap influences the performance of our method. The more edges in the roadmap, the more probes need to be sent and, hence, the interval tree grows wider. On the other hand, more possibilities become available in the roadmap when the roadmap gets larger. This can also have a positive effect on the performance, as this may lower the delay factor of the problem.

To avoid strong deviations in the running times of the experiments as a result of the randomness involved in creating roadmaps, we extend roadmaps over the experiments. To be precise, in each experiment, we add 100 vertices to the roadmap of the previous experiment. The number of edges grows more or less proportionally. The results are shown in Fig. 15 and Table I.

The figure shows a linear relation between the roadmap size and the running time. We see that for a too-small roadmap, the running time actually gets higher. This is because the possibilities of the robot are too much restricted, which results in a high delay factor of the problem (see Table I). The minimum running time is found for a roadmap with 200 vertices (and 558 edges). Note that the resulting trajectory is not substantially longer than the trajectories found in larger roadmaps (see column "trajectory length").

TABLE I
RESULTS FOR VARIOUS ROADMAP SIZES

# vertices	# edges	roadmap distance	trajectory length	delay-factor	running time
100	246	17.15	43.40	2.53	0.67s
200	558	16.52	25.58	1.55	0.33s
300	944	16.52	25.48	1.54	0.62s
400	1376	16.52	25.48	1.54	1.16s
500	1806	14.70	25.48	1.73	1.88s
600	2236	14.70	25.48	1.73	2.13s
700	2698	14.70	25.48	1.73	2.48s
800	3156	14.70	25.48	1.73	2.85s
900	3668	14.21	24.64	1.73	3.17s
1000	4158	14.21	24.64	1.73	3.90s

Time Step: The effect of the value of the time step Δt is inversely proportional to the running time of our method. That is, as $1/\Delta t$ increases, the running time linearly grows accordingly. This is confirmed by the results of experiments we performed for several values of Δt.

Other Quantities: Other quantities do not influence the performance of our method. The number of DOFs of the robot, for instance, or the size of the environment, only influence the performance if they make the problem more difficult, or if they make the environment require a larger roadmap. Also, some quantities, such as the number of dynamic obstacles or the complexity of the dynamic obstacles, cause collision checks to become more expensive. They may as such only have an indirect relation to the performance of our method.

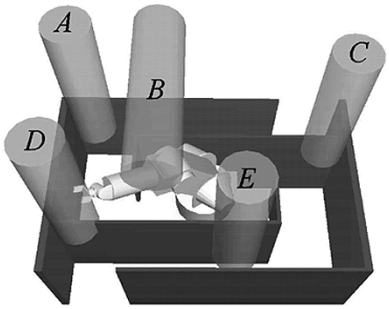

Fig. 16. Dynamic environment with an articulated robot and five dynamic obstacles. Obstacles A, B, and E move back and forth parallel to the x axis. C and D move parallel to the y axis.

C. Articulated Robots

Our method is suitable for articulated robots as well. A particular advantage of our method with respect to articulated robots is that no expensive checks for self-collisions have to be done during the algorithm. These are already performed during the preprocessing phase.

We performed an experiment involving an articulated robot with six DOFs. The environment in which the experiment is performed is depicted in Fig. 16. The articulated robot stands fixed amidst some static walls. Five dynamic obstacles (cylinders) hinder the robot in its attempt to go from a start configuration (robot fully bent to the right) to the goal configuration (robot fully bent to the left). Although it is not a very realistic environment, it shows the capabilities of our method.

A roadmap was created for the articulated robot using the variant of PRM that allows cycles in the roadmap. The construction stopped when the start and goal configuration were connected by the roadmap. A roadmap containing only 20 vertices and 49 edges proved to be enough to cover this rather simple static scene. The shortest path in the roadmap takes 3.78 units of time to traverse if there are no dynamic obstacles. The trajectory avoiding the dynamic obstacles computed by our method takes 7.86 units of time to traverse. Hence, the delay factor for this problem is 2.08. The value of Δt was chosen adequately small. It took only 0.87 s to compute the trajectory using our method.

We must note that the collision checks were more expensive for this environment than for the previous one. On the one side, this is because the number of dynamic obstacles is larger, but more significant is that it takes much more time to configure and collision check an articulated robot than a free-flying robot.

D. Multirobot Motion Planning

One of the applications of motion planning in dynamic environments is motion planning for multiple robots. In such problems, the simultaneous motion of a set of robots is considered. Each robot has its own start and goal configuration, and each robot has to reach its goal configuration without collisions with static obstacles or other robots. Furthermore, a performance measure is defined that should be optimized. A common aim is to minimize the time it takes for the last robot to reach its goal, with a secondary objective to optimize the arrival time of each individual robot.

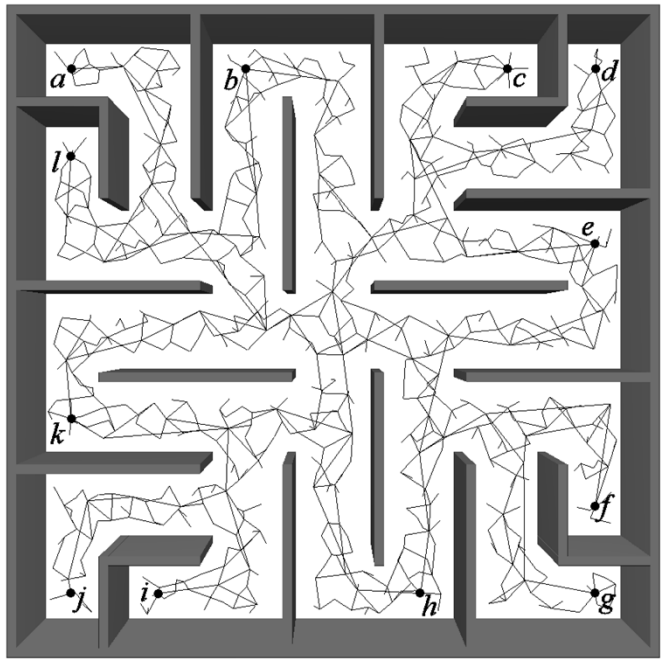

Fig. 17. Static environment and its roadmap with 12 query configurations (a to l).

One out of many approaches to motion planning for multiple robots is *prioritized planning* [2]. It works as follows. First, priorities are assigned to the robots. Then in order of decreasing priority, the robots are picked. For each picked robot, a trajectory is planned, avoiding collisions with the static obstacles as well as the previously picked robots, which are considered as dynamic obstacles. Our method is perfectly suited for this scheme, provided that for each robot individually a roadmap is constructed for the static part of the scene.

An important question is how to assign the different priorities to the robots. For our purpose a simple heuristic suffices: the priority of a robot equals the roadmap distance between its start and goal vertex. The rationale behind this heuristic is, keeping in mind that we aim to minimize the arrival time of the last robot, that robots that have to traverse long distances (and hence, need much time) can do this relatively unhindered, while robots that have to traverse short distances can afford to spend time on avoiding robots with higher priority.

The combination of the above scheme with our method allows for a broad applicability. There is, for instance, no limitation on the number of DOFs of each of the robots. They can be of different types, for example, articulated and free-flying robots can be used simultaneously. Also, the scheme allows for motion planning for multiple robots in dynamic environments.

We experimented with our method in the environment of Fig. 17. All the robots are cylinders that can translate in the plane. Since all the robots are the same, they can use the same roadmap. It consists of 750 vertices and 1004 edges (see Fig. 17).

In the roadmap, we have defined 12 query configurations (a to l). Our first experiment comprehends six robots with the queries $a \rightarrow g, b \rightarrow h, c \rightarrow i, d \rightarrow j, e \rightarrow k$, and $f \rightarrow l$. The environment and the queries are chosen such that a clash might arise in the center of the environment. All the shortest paths

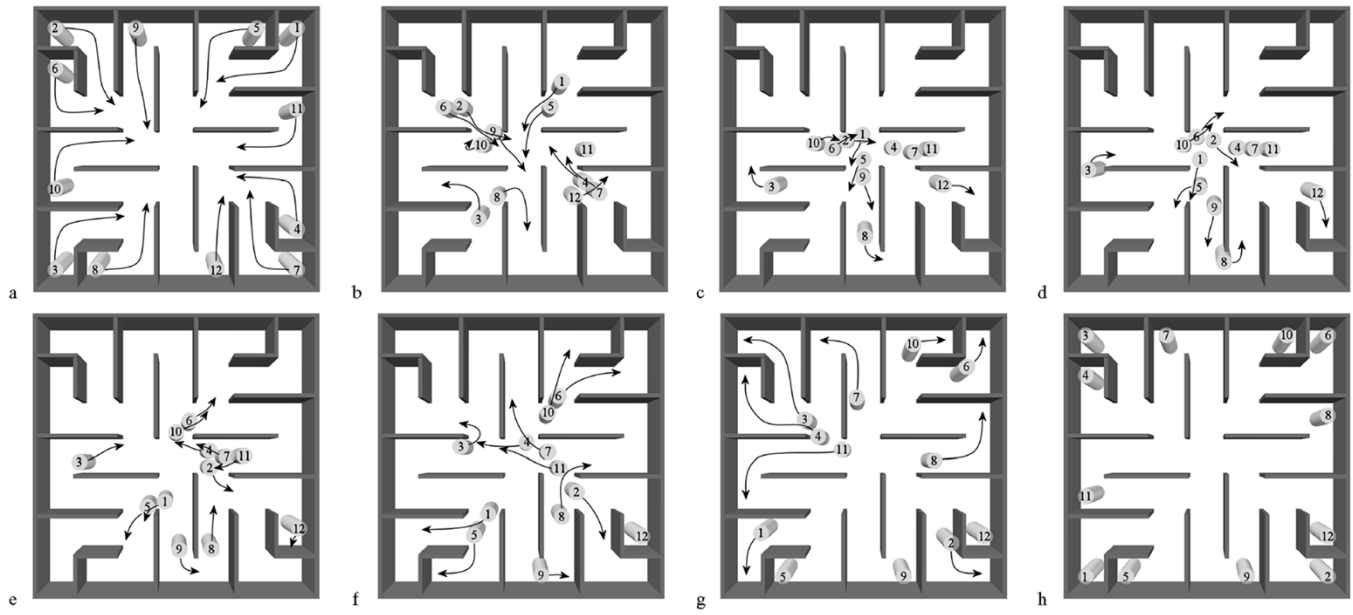

Fig. 18. Pictures from the multirobot trajectory. The start and goal configurations are shown in (a) and (h), respectively. Each robot is labeled according to its rank. The robots clash in the center (c). Only robots 3 and 8 choose to take a detour and avoid the bustle.

TABLE II
RESULTS FOR SIX ROBOTS

robot rank	query	roadmap distance	trajectory length	delay-factor	running time
1	$d \rightarrow j$	11.48	11.48	1.00	0.01s
2	$a \rightarrow g$	11.13	11.55	1.04	0.01s
3	$f \rightarrow l$	10.36	12.88	1.24	0.09s
4	$c \rightarrow i$	9.94	13.23	1.33	0.17s
5	$b \rightarrow h$	8.33	8.33	1.00	0.01s
6	$e \rightarrow k$	7.91	10.78	1.36	0.17s
total		11.48	13.23		0.46s

TABLE III
RESULTS FOR 12 ROBOTS

robot rank	query	roadmap distance	trajectory length	delay-factor	running time
1	$d \rightarrow j$	11.48	11.48	1.00	0.01s
2	$a \rightarrow g$	11.13	11.55	1.04	0.01s
3	$j \rightarrow a$	10.78	13.23	1.23	0.11s
4	$f \rightarrow l$	10.36	13.37	1.29	0.17s
5	$c \rightarrow i$	9.94	9.94	1.00	0.01s
6	$l \rightarrow d$	9.80	11.20	1.14	0.06s
7	$g \rightarrow b$	9.66	12.67	1.31	0.24s
8	$i \rightarrow e$	9.45	12.46	1.32	0.30s
9	$b \rightarrow h$	8.33	8.54	1.03	0.02s
10	$k \rightarrow c$	8.26	10.99	1.33	0.27s
11	$e \rightarrow k$	7.91	13.51	1.71	0.61s
12	$h \rightarrow f$	5.18	7.21	1.39	0.13s
total		11.48	13.51		1.94s

between the query configurations go through this center region. There are detours possible that avoid the center, but they are much longer.

To compute a multirobot trajectory, we ordered the queries of the robots according to their roadmap distance, and executed our method on each of the robots in this order. The dynamic environment in each iteration consists of the trajectories previously computed for the robots with a higher rank. For the robot with the highest rank, the dynamic environment is empty, so its trajectory is returned instantly.

The results of the experiment are given in Table II. Our scheme was able to compute a multirobot trajectory for six robots in only 0.46 s of running time. The arrival time of the last robot is 13.23, where, in any case, 11.48 units of time would have been needed to complete a trajectory of the multirobot.

In the second experiment, we added six more robots with queries $g \rightarrow b, h \rightarrow f, i \rightarrow e, j \rightarrow a, k \rightarrow c$, and $l \rightarrow d$, to make a total of as many as 12 robots. The results are shown in Table III. The multirobot trajectory was computed in only 1.94 s, and the arrival time of the last robot is only 13.51 (compare with the previous experiment, where this was 13.23). The trajectory is visualized in Fig. 18.

In Tables II and III, we see that the delay factors are not very high, despite the fact that the environment gets quite crowded with 12 robots. We expect that this is exemplary, and that in practical situations, the delay factor will rarely exceed a value of two. Yet, as we saw in the first two experiments, our method works fine, even for high delay factors.

E. Comparison With a Brute-Force Method

To adequately assess the performance of our method, we compared our method with a "brute-force" method. The brute-force method simply performs an A* search in the space formed by the Cartesian product of the discretized roadmap and the discretized time axis. It finds the same trajectories as our method, i.e., near-time-optimal ones. We ran the brute-force method on the examples discussed in this paper, and the results show that the brute-force method is, on average, at least a factor 10 slower than our method. Also, it is more sensitive to changes in the size of the roadmap.

VIII. 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 DOFs.

An application of particular interest is motion planning for multiple robots. Our experiments show that complicated planning problems for as many as 12 robots can be solved in less than 2 s of computation time.

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

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 two-level strategy of the algorithm. We exploit the advantages of depth-first search on local trajectories, and use A* to find a global trajectory. Only the potentially interesting parts of the implicit interval tree are evaluated. Note that if no moving obstacles are present, only probes along the path in the roadmap leading directly to the goal vertex 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.

The current algorithm finds trajectories reaching the goal vertex as soon as possible. An interesting extension to our method is to be able to compute trajectories arriving on the goal vertex at a specific moment in time. In principle, such a change is easily implemented, but then the method loses its efficiency because the A* search is wrongly focussed. Changing the A* focus appears not to be trivial, so this remains an interesting subject of further research.

ACKNOWLEDGMENT

The authors would like to thank D. Nieuwenhuisen and R. Geraerts for codeveloping the experimentation software.

REFERENCES

[1] B. Baginski, "The Z^3-method for fast path planning in dynamic environments," in *Proc. IASTED Conf. Applicat. Control Robot.*, 1996, pp. 47–52.

[2] M. Erdmann and T. Lozano-Pérez, "On multiple moving objects," *Algorithmica*, vol. 2, pp. 477–521, 1987.

[3] P. Fiorini and Z. Shiller, "Time optimal trajectory planning in dynamic environments," *J. Appl. Math. Comput. Sci.*, vol. 7, no. 2, pp. 101–126, 1997.

[4] ——, "Motion planning in dynamic environments using velocity obstacles," *Int. J. Robot. Res.*, vol. 17, no. 7, pp. 760–772, 1998.

[5] T. Fraichard, "Trajectory planning in a dynamic workspace: A "state-time" approach," *Adv. Robot.*, vol. 13, no. 1, pp. 75–94, 1999.

[6] K. Fujimura, *Motion Planning in Dynamic Environments*. Tokyo, Japan: Springer-Verlag, 1991.

[7] ——, "Time-minimum routes in time-dependent networks," *IEEE Trans. Robot. Autom.*, vol. 11, no. 3, pp. 343–351, Jun. 1995.

[8] D. Hsu, R. Kindel, J.-C. Latombe, and S. Rock, "Randomized kinodynamic motion planning with moving obstacles," *Int. J. Robot. Res.*, vol. 21, no. 3, pp. 233–255, 2002.

[9] L. Kavraki, P. Švestka, J.-C. Latombe, and M. H. Overmars, "Probabilistic roadmaps for path planning in high-dimensional configuration spaces," *IEEE Trans. Robot. Autom.*, vol. 12, no. 4, pp. 566–580, Aug. 1996.

[10] J.-C. Latombe, *Robot Motion Planning*. Boston, MA: Kluwer, 1991.

[11] S. M. LaValle and J. J. Kuffner, Jr., "Randomized kinodynamic planning," in *Proc. IEEE Int. Conf. Robot. Autom.*, 1999, pp. 473–479.

[12] D. Nieuwenhuisen and M. H. Overmars, "Useful cycles in probabilistic roadmap graphs," in *Proc. IEEE Int. Conf. Robot. Autom.*, 2004, pp. 446–452.

[13] D. Nieuwenhuisen, A. Kamphuis, M. Mooijekind, and M. H. Overmars, "Automatic construction of roadmaps for path planning in games," in *Proc. Int. Conf. Comput. Games: Artif. Intell., Design, Educ.*, 2004, pp. 285–292.

[14] J. Reif and M. Sharir, "Motion planning in the presence of moving obstacles," in *Proc. IEEE Found. Comput. Sci.*, 1985, pp. 144–154.

[15] P. Švestka, "Robot motion planning using probabilistic road maps," Ph.D. dissertation, Utrecht Univ., Utrecht, The Netherlands, 1997.

Jur P. van den Berg received the M.Sc. degree in computer science from the University of Groningen, Groningen, The Netherlands, in 2003. He is currently working toward the Ph.D. degree at Utrecht University, Utrecht, The Netherlands.

His main research interests are robotics and computational geometry, with a particular focus on motion planning.

Mark H. Overmars received the Ph.D. degree in computer science in 1983 from Utrecht University, Utrecht, The Netherlands.

Currently he is a full Professor with the Department of Computer Science at the same university, where he also heads the Center for Geometry, Imaging, and Virtual Environments. His main research interests include computational geometry and its application in areas including virtual reality, robotics, and computer games.