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

Kinodynamic Motion Planning on Roadmaps in Dynamic Environments

Jur van den Berg Mark Overmars

***Abstract*— In this paper we present a new method for kinodynamic motion planning in environments that contain both static and moving obstacles. We present an efficient two-stage approach: in the preprocessing phase, it constructs a roadmap that is collision-free with respect to the static obstacles and encodes the kinematic constraints on the robot. In the query phase, it plans a time-optimal path on the roadmap that obeys the dynamic constraints (bounded acceleration, curvature derivative) on the robot and avoids collisions with any of the moving obstacles. We do not put any constraints on the motions of the moving obstacles, but we assume that they are completely known when a query is performed. We implemented our method, and experiments confirm its good performance.**

I. Introduction

The problem we discuss in this paper is kinodynamic motion planning in dynamic environments. That is, planning a path for a robot from a start to a goal state in a two- or three-dimensional workspace that obeys the kinematic and dynamic constraints on the robot and avoids collisions with static and moving obstacles in the environment. We assume that the geometry and motions of the obstacles are given.

The basic, static motion planning problem is usually formulated in terms of the *configuration space*, of which each dimension corresponds to a degree of freedom of the robot. To take the dynamics of the robot into account, the configuration space is extended to the *state space*, whose *states* not only contain information about the configuration of the robot, but also about its velocity. Planning in state spaces is harder than planning in configuration spaces, as, for instance, a straight line through the state space is not a valid motion in general [14].

The approaches that have been suggested for kinodynamic planning can roughly be divided into two categories. The first *discretizes* the state space into a regular grid of reachable states [5], and the second *randomly* grows a tree of valid paths from the start state until a goal region is reached [13].

For dynamic environments, the notion of state-space is extended to the *state-time space*, in which time is included as an additional dimension. Approaches for planning in these spaces are based on the two sets of approaches mentioned above. The methods of [8], [17] build a tree of valid motions from the start state for each planning query. New branches in the tree are generated by randomly sampling an action from the *control space* of the robot, and integrating it over a short period of time. Although some promising results have been achieved, a major drawback is that narrow passages caused by static obstacles may (heavily) burden the running time of the method, because all the effort is done in the query phase. Further, it is only possible to reach a goal region, rather than a specified goal state, as the planned paths are sequences of random actions. This latter fact may also cause the paths to look rather unnatural.

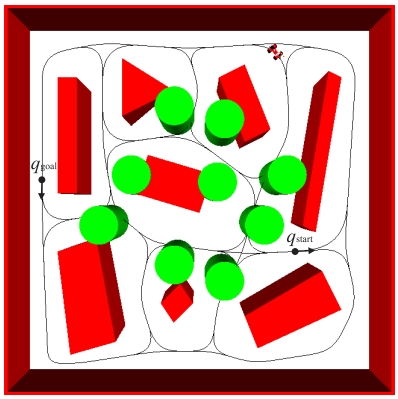

Fig. 1. An environment with 9 moving obstacles (cylinders). The robot is shown in the upper-right corner.

The method of [6] discretizes the state-time space into a grid, and searches it for an optimal motion. However, to keep the method computationally feasible, the robot is constrained to move over a pre-planned *path*, that is collision-free with respect to the static obstacles. A state then encodes the robot's position and velocity along the path, resulting in a three-dimensional state-time space. The major drawback of this method is that constraining the robot's motion to a path substantially decreases the maneuverability of the robot. The method presented in [1] does not have this drawback, as it constrains the robot to a *roadmap*, but this method is not able to take any dynamic constraints on the robot into account.

In this paper, we propose a new approach that combines the good properties of the latter two methods. It uses a preprocessed roadmap to guide the motion of the robot, giving a large maneuverability while keeping the planning problem tractable, and it also takes the kinematic and dynamic constraints on the robot into account.

The roadmap is built in a preprocessing phase, such that it is collision-free with respect to the static obstacles and encodes the kinematic constraints on the robot. Hence, a roadmap has to be built specifically for the particular kinematics model of the used robot. In this paper we choose the car-like robot as example, but the results apply to other

This research was supported by the Dutch BSIU/BRICKS project and by the GATE project, funded by the Netherlands Organization for Scientific Research (NWO) and the Netherlands ICT Research and Innovation Authority (ICT Regie).

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

Mark Overmars is with the Department of Information and Computing Sciences, Universiteit Utrecht, The Netherlands (e-mail: markov@cs.uu.nl).

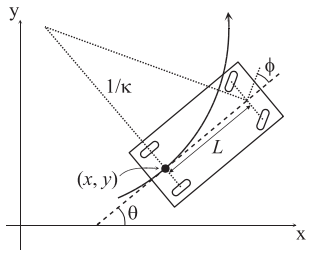

Fig. 2. The kinematic model of a car-like robot.

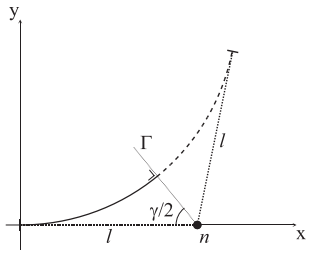

Fig. 3. Creating a shortcut between two edges in the roadmap.

motion models as well (e.g. human walk [16]). As the quality of the paths largely depends on the quality of the roadmap, we require a smooth roadmap containing natural and feasible paths (see Fig. 1). In Section II we introduce the motion model for car-like robots and in Section III we will show how we construct such roadmaps.

Given such a roadmap, a start and goal state on the roadmap, and a start time, our method plans an approximately *time-optimal* path on the roadmap that obeys the dynamic constraints on the robot and avoids collisions with any of the moving obstacles. To this end, time is discretized into small steps, and at each time step the robot either accelerates maximally, maintains its current velocity, or decelerates maximally. This discretizes the set of possible velocities and the set of possible positions as well. Hence, we will plan in three-dimensional *state-time grids* along the edges of the roadmap. In [6], a standard A*-algorithm was used to plan through such grids, but we will introduce a more efficient search algorithm here. We do not put any constraints on the motions of the moving obstacles, but we assume that they are completely known when a query is performed. Experiments show that our algorithm is capable of planning in complex environments in the order of a second of running time. The algorithm will be presented in detail in Section IV and in Section V we show our experimental results. We conclude the paper in Section VI.

II. Car-Like Robots

The results we present in this paper apply to different kinds of kinematic models. For ease of presentation though, we focus on the car-like model. It is an intuitive model that is often used in literature [14].

A. Kinematic Model of a Car-Like Robot

A car can be imagined as a rectangle moving in the plane. Its configuration is defined by a position (x, y), and an orientation θ (see Fig. 2). Let L be the distance between the rear axle and the front axle of the car. The configuration transition equations of the car, in terms of the *path length* s, are given by:

$$x'(s) = \cos\theta \quad (1)$$

$$y'(s) = \sin\theta \quad (2)$$

$$\theta'(s) = \tfrac{1}{L}\tan\phi = \kappa, \quad (3)$$

where ϕ is the car's steering angle, and κ the curvature of the followed path. To have realistic motions, it is important that the steering angle ϕ is continuous along the robot's paths. This is achieved when the derivative of the curvature is constant, say K. Hence, $\kappa'(s) = K$. Integrating the above equations then gives the following:

$$\theta(s) = \int \kappa(s)\, ds = \tfrac{1}{2}Ks^2 \quad (4)$$

$$x(s) = \int \cos\theta(s)\, ds = \sqrt{\tfrac{\pi}{K}}C\big(\sqrt{\tfrac{K}{\pi}}s\big) \quad (5)$$

$$y(s) = \int \sin\theta(s)\, ds = \sqrt{\tfrac{\pi}{K}}S\big(\sqrt{\tfrac{K}{\pi}}s\big), \quad (6)$$

where $C(\cdot)$ and $S(\cdot)$ are the Fresnel integral functions. The curves described by these equations are called *clothoids*.

B. Dynamic Constraints

Above we have seen what curves the robot will traverse. Here we give the model for the motions along these curves with respect to time t, taking into account the dynamic constraints on the velocity and acceleration of the robot. This gives the following set of equations:

$$s'(t) = v \quad (7)$$

$$v'(t) = a, \quad (8)$$

where v is the velocity of the robot, and a its acceleration. We bound the velocity and acceleration from above and below: $v_{\min} \le v \le v_{\max}$, and $-a_{\max} \le a \le a_{\max}$. Note that $v_{\min} \le 0$, and that a negative velocity corresponds to moving *backwards*. Because of the discretization we apply later, we choose symmetric bounds on the acceleration.

To have realistic motions, we bound the speed with which the steering wheel can be turned. That is, the absolute value of the derivative to time of the steering angle ϕ is upper-bounded by some maximum Φ. This is achieved when the derivative to time of the curvature κ is upper-bounded by $\frac{\Phi}{L}$:

$$|\kappa'(t)| \le \Phi/L. \quad (9)$$

So, when we are given a clothoid curve for some value of K (see Equations (5) and (6)), the maximal velocity with which it can be traversed is $\frac{\Phi}{|K|L}$, in order to obey the above constraint. We will use this constraint as an example in this paper, but additional constraints may be introduced on the curvature derivative to time, for instance to prevent the car from slipping away in curves (see, e.g., [6]).

III. Creating Roadmaps for Car-Like Robots

In [18], a method is presented that creates a roadmap for the above kinematic model, based on the well known PRM-method [9]. It randomly samples configurations (x, y, θ), and connects them by paths having a fixed curvature derivative. Hence, their method is mainly suitable in situations where the robot has a fixed velocity. Also, their roadmaps may contain unnatural paths due to the randomness involved in the creation process.

For that reason, we take a slightly different approach for creating roadmaps: we start with a straightforward roadmap containing nodes connected by straight-line edges, and then shortcut each of the sharp turns in the roadmap by a clothoid

curve (a similar approach has been proposed in [15], but it uses circular arcs as shortcuts, which do not give smooth steering wheel motions). We want the shortcuts to have the least possible curvature derivative, such that they can be traversed with the largest possible velocity. Therefore, we prefer an input roadmap having a small number of long edges that cover the connectivity of the free configuration space well, and have some clearance from the static obstacles. As we use simple roadmaps as input, we can exploit the vast amount of literature on the topic of creating them (see, e.g., [7] for creating small, well covering roadmaps).

A. Computing Shortcuts

For every pair of edges in the simple roadmap that are incident to a common node, we compute a shortcut curve and add it to the new roadmap. Let us look at a node n with a pair of incident edges making angle γ. Let ℓ be the minimum of the half lengths of both edges. We compute a curve smoothly connecting the points lying at distance ℓ from n along both of the edges (in order to have a symmetric curve). Without loss of generality, we rotate and translate the edges such that node n lies at $(\ell, 0)$ and that one of its incident edges lies on the x-axis (see Fig. 3). Let Γ be the line supporting the bisector of angle γ. The challenge is to find a value for K, such that the clothoid curve defined by Equations (5) and (6) intersects Γ perpendicularly. This gives the following system of equations (note that Γ is defined by the equation $x = \ell - \tan(\frac{\pi}{2} - \frac{\gamma}{2})y$):

$$\theta(\hat{s}) = \frac{\pi}{2} - \frac{\gamma}{2} \quad (10)$$

$$x(\hat{s}) = \ell - \tan(\tfrac{\pi}{2} - \tfrac{\gamma}{2})y(\hat{s}), \quad (11)$$

where $\hat{s}$ is the s-value for which the clothoid intersects Γ. Solving for $\hat{s}$ and K defines the part of the shortcut left of Γ. It is given by Equations (5) and (6) for $0 \leq s \leq \hat{s}$. The right part is obtained by mirroring the left part in Γ.

If the shortcut is collision-free with respect to the static obstacles, we can add it to the new roadmap. If not, we can recompute a shortcut for $\ell \leftarrow \ell/2$. As the simple roadmap is collision-free, we are guaranteed to find a collision-free shortcut eventually as ℓ approaches zero. In Fig. 4 an example is given of a simple input roadmap and the smooth roadmap suitable for car-like robots that is obtained after adding the shortcuts. Creating the roadmap took 0.07s on an Intel Core2-6400, 2.13GHz. As can be seen from the figure, the roadmap contains natural paths, and its shape is easily controllable by manipulating the input roadmap. The roadmap is valid for cars with any value of L, as far as kinematics are concerned.

The roadmap resulting from the above procedure is symmetric, that is, it can be traversed in both directions, but for convenience we duplicate each of the edges in the roadmap to form *directed* edges, and connect them such that sharp turns can *not* be taken at the vertices where three (or more) edges of the roadmap come together. As a result, any path in the roadmap is indeed a valid motion for the car-like robot. By using directed edges, we also distinguish between moving backwards, and moving forward in the opposite direction, which are different motions, obviously.

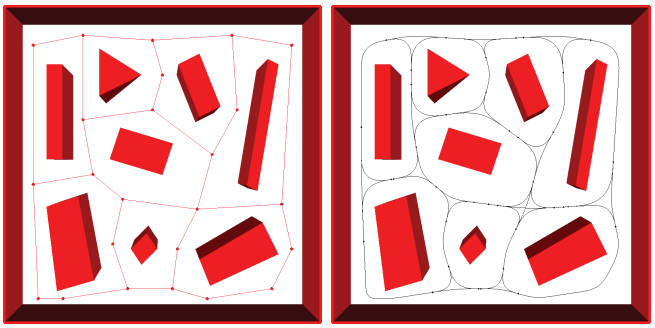

Fig. 4. A simple roadmap having straight-line edges (left), and the smooth roadmap after shortcutting all nodes by clothoid curves (right).

IV. Planning among Moving Obstacles

Above, we have seen how to create a roadmap that encodes the kinematic constraints on the robot, and is collision-free with respect to the static obstacles in the environment. To obey the dynamic constraints on the robot, we have to consider the *state space* of the robot, which we will introduce below. To avoid the moving obstacles in the environment, we extend the state space to the *state-time space*, which we will introduce in Section IV-B. An efficient algorithm to plan a time-optimal path on the roadmap between a given start and goal state is presented in Section IV-C. We assume that the start and goal state are contained in the roadmap (this can always be achieved by connecting them to the roadmap before performing the query).

A. State Space

Let us first assume that the roadmap consists of a single path. The *state space* of the robot then consists of pairs $\langle s, v \rangle$, where s is the position of the robot along the path, and v the robot's velocity. We discretize the state space into a grid by choosing a small time step Δt. At each time step, the robot is allowed to either accelerate maximally, maintain its current velocity, or decelerate maximally. This gives the following state transition equations:

$$a \in \{-a_{\max}, 0, a_{\max}\} \quad (12)$$

$$v(t + \Delta t) = v(t) + a\Delta t \quad (13)$$

$$s(t + \Delta t) = s(t) + v(t)\Delta t + \tfrac{1}{2}a\Delta t^2. \quad (14)$$

They result in a regular two-dimensional grid of reachable states (see Fig. 5), where the spacings in the grid are $\Delta v = a_{\max}\Delta t$ along the v-axis, and $\Delta s = \frac{1}{2}a_{\max}\Delta t^2$ along the s-axis. From a given state $\langle s, v \rangle$, three other states are reachable: $\langle s + (2\frac{v}{\Delta v} + 1)\Delta s, v + \Delta v \rangle$, $\langle s + 2\frac{v}{\Delta v}\Delta s, v \rangle$ and $\langle s + (2\frac{v}{\Delta v} - 1)\Delta s, v - \Delta v \rangle$, each one associated with a different acceleration. This defines a directed graph in the discretized state space which we call the *state graph*.

To create the state graph for an entire roadmap rather than a single path, we construct a state grid along each of the edges of the roadmap and connect them at the vertices of the roadmap, such that the robot can choose among all of

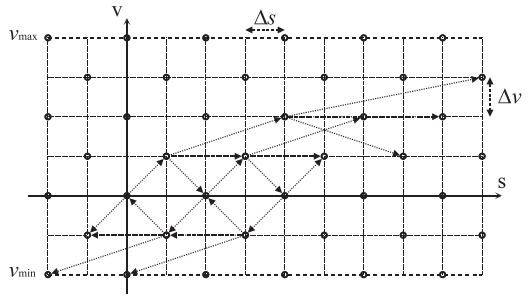

Fig. 5. The state grid along a single edge of the roadmap. Only the grid points marked by the dots are reachable. A part of the state graph is shown using dotted arrows. Each transition takes one time step.

the outgoing edges when it encounters a vertex. As can be seen in Fig. 5, only half of the states in the state grid are reachable. So, in order to connect the state grids smoothly at the vertices, we subdivide each of the edges of the roadmap into steps of the largest possible length smaller than Δs, such that the edge is subdivided exactly into an *even* number of steps. As a result, there is a finite number of reachable positions in the roadmap. For each of these positions, the velocity is bounded from above and below; either by $v_{\max}$ and $v_{\min}$, or by the dynamic constraint of Equation (9). Hence, the total state graph contains a finite number of states. We construct this state graph *explicitly*.

In the experiments we performed in the roadmap of Fig. 4 (the scene has dimensions 60×60), we chose $\Delta t = 0.2$ and $a_{\max} = 5$, resulting in $\Delta v = 1$ and $\Delta s = 0.1$. Further, we chose $v_{\max} = 10$, $v_{\min} = -2$, $L = 2$ and $\Phi = 1$, which resulted in a state graph containing 75,609 valid states. Constructing it took 0.50 seconds, but as it is part of the preprocessing, this step is not time-critical.

B. State-Time Space

To plan over the roadmap while avoiding collisions with the moving obstacles, we add the time dimension to the discretized state space, forming a three-dimensional *state-time space* along each of the edges of the roadmap (see Fig. 6). It consists of pairs $\langle q, t\rangle$, where $q = \langle s, v\rangle$ is a state contained in the state graph, and t a time value. The time axis is discretized by the time step Δt. The moving obstacles in the environment transform to static obstacles in the state-time space. They are *cylindrical* along the v-dimension, as the robot's velocity does not influence its collision status.

Like we defined the state graph on the discretized state space, we define the *state-time graph* on the discretized state-time space. It is a directed *acyclic* graph, that contains a transition from state-time $\langle q, t\rangle$ to $\langle q', t + \Delta t\rangle$ if q' is a successor of q in the state graph. Also, the transition needs to be *collision-free* with respect to the moving obstacles. As its length (in terms of time) is only Δt, we assume that this is the case when both of the state-times it connects are collision-free (such an approximation is common in motion planning [1], [12]).

The task is to plan a collision-free path through the state-time graph from a given start state-time $\langle q_{\text{start}}, t_{\text{start}}\rangle$ that reaches a given goal state q_{goal} as soon as possible, i.e. for the lowest possible time value. Unlike the state graph, the state-time graph is infinite, so we construct it *implicitly* during the search for a valid path.

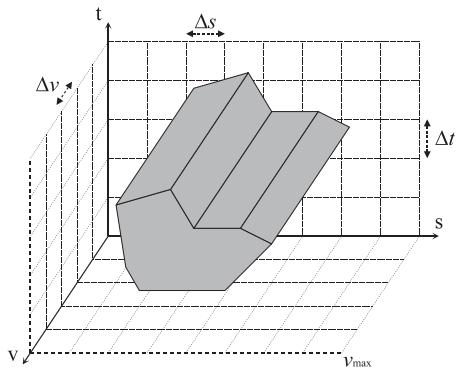

Fig. 6. The three-dimensional state-time grid along a single edge of the roadmap. Obstacles (grey) are cylindrical along the v-dimension.

C. Planning Algorithm

A straightforward approach for searching a time-optimal path is the A*-algorithm. It builds a *shortest path tree* rooted at the start state-time and biases its growth towards the goal. To this end, A* maintains the leafs of the tree in a priority queue $\mathcal{Q}$, and sorts them according to their f-value. The function $f(\langle q, t\rangle)$ gives an estimate of the length (in terms of time) of the shortest path from the start to the goal via $\langle q, t\rangle$. It is computed as $g(\langle q, t\rangle) + h(\langle q, t\rangle)$ where $g(\langle q, t\rangle)$ is the time it takes to go from the start to $\langle q, t\rangle$, and $h(\langle q, t\rangle)$ a lower-bound estimate of the time it takes to reach the goal from $\langle q, t\rangle$. In our case, $g(\langle q, t\rangle) = t$ and $h(\langle q, t\rangle)$ is the distance in the state graph between state q and the goal state q_{goal}. This distance is acquired for all states by performing a single backwards *breadth-first search* on the state graph from q_{goal} prior to executing the A*-algorithm. A* is initialized with the start state-time in its priority queue, and in each iteration it takes the state-time from the queue with the lowest f-value and checks it for collisions. If it is collision-free, each successor of this state-time that has not been visited before is inserted into the queue. This process repeats until the goal state is reached. The algorithm is given below.

Algorithm 1 A*$(q_{\text{start}}, t_{\text{start}}, q_{\text{goal}})$

```
Insert ⟨q_start, t_start⟩ into Q
while Q is not empty do
    Pop the element ⟨q, t⟩ with lowest f-value from Q
    if ⟨q, t⟩ is collision-free then
        if q = q_goal then return success!
        for all successors q' of q in the state graph do
            if not ⟨q', t + Δt⟩.visited then
                ⟨q', t + Δt⟩.backpointer ← ⟨q, t⟩
                ⟨q', t + Δt⟩.visited ← true
                Insert ⟨q', t + Δt⟩ into Q
Path does not exist; return failure
```

The state-time graph differs considerably in nature from regular (directed acyclic) graphs. This is because of each state-time $\langle q, t\rangle$ it is clear a priori that *if* it can be reached,

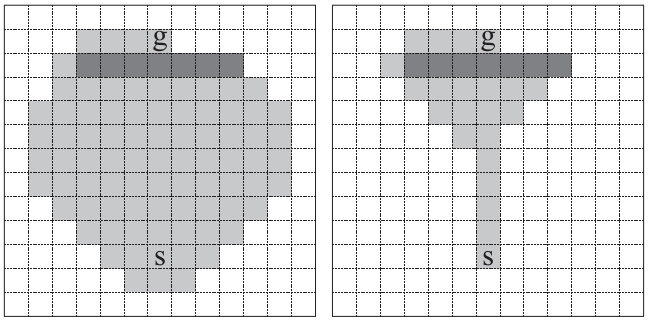

Fig. 7. The states collision-checked (gray) by an A*-search (left) and a repeated A*-search using the freespace assumption (right) in an implicit 8-connected grid where each traversal costs 1. The dark states are colliding.

it will be reached in t time. Hence, we do not need to *relax* state-times in the algorithm. Therefore, the above algorithm is not truly an A*-algorithm, but rather a *prioritized breadth-first search*. Also, the state-time graph is *implicit*; whether or not traversing a state-time is possible (due to a collision) has to be checked "on demand" during the exploration of the graph. Note that collision-checking is an expensive operation in general, so the cylindrical structure of the state-time obstacles is exploited to save collision-checks by reusing the result of previous collision-checks.

We can do better than Algorithm 1, though. If one would only be concerned with minimizing the number of collision-checks, an approach taking the *freespace assumption* [10] is cheaper. It assumes that the state-times are traversable unless it knows otherwise. So, it repeatedly plans a time-optimal path from start to goal, and then collision-checks this path. If a state-time along the path appears to be untraversable, a new path is planned taking the newly acquired information into account. This process is repeated until a path is found that is fully collision-free from start to goal. If the backpointers are maintained such that collision-checks from previous paths are reused maximally, this scheme is guaranteed to use less or equal collision-checks than Algorithm 1. In Fig. 7, the difference in the number of collision checks is shown for a simple 8-connected grid.

This procedure, however, introduces the overhead of repeatedly rebuilding a shortest path tree. This can be remedied by letting the repeated searches maintain a single shortest path tree, similar to Lifelong Planning A* [11]. This method was developed for navigation in unknown terrain, but we adapted it for efficient planning in the implicit state-time graph. The algorithm is shown in Algorithm 2 (its overhead can be reduced further, but we left out some details).

The algorithm repeatedly plans a path to the goal given the current collision-check information (lines 3-11). When a path has been found, it is collision-checked from the start towards the goal, and if a collision is detected, the algorithm attempts to connect the branch of the shortest path tree emanating from the invalidated state-time to other, valid state-times. If this fails, it is erased and its leafs whose corresponding state-times are in the priority queue are removed from the queue (lines 17-25). This process is repeated until a path is found that is completely collision-free (line 15).

Algorithm 2 REPEATEDA*$(q_{\text{start}}, t_{\text{start}}, q_{\text{goal}})$

```
 1: Insert ⟨q_start, t_start⟩ into Q
 2: while true do
 3:    ⟨q, t⟩ ← element of Q with lowest f-value
 4:    while Q is not empty and q ≠ q_goal do
 5:       Remove ⟨q, t⟩ from Q
 6:       for all successors q' of q in the state graph do
 7:          if not ⟨q', t + Δt⟩.visited then
 8:             ⟨q', t + Δt⟩.backpointer ← ⟨q, t⟩
 9:             ⟨q', t + Δt⟩.visited ← true
10:             Insert ⟨q', t + Δt⟩ into Q
11:       ⟨q, t⟩ ← element of Q with lowest f-value
12:    if Q is empty then path does not exist; return failure
13:    Construct path π from start to goal by following the back-
       pointers from ⟨q_goal, t⟩
14:    Collision-check state-times in π in order of increasing t-
       value and stop when a collision is encountered. Let ⟨q, t⟩
       be the last checked state-time
15:    if no collision was encountered in π then return success!
16:    if q = q_goal then remove ⟨q, t⟩ from Q
17:    else put all successors of ⟨q, t⟩ into a FIFO-queue F
18:    while F not empty do
19:       Pop front element ⟨q', t'⟩ from F
20:       if ⟨q', t'⟩ has a visited valid predecessor then
21:          ⟨q', t'⟩.backpointer ← predecessor with lowest f-value
22:       else
23:          ⟨q', t'⟩.visited ← false
24:          if ⟨q', t'⟩ ∈ Q then remove ⟨q', t'⟩ from Q
25:          else put all successors of ⟨q', t'⟩ into F
```

V. EXPERIMENTAL RESULTS

We implemented both Algorithm 1 and Algorithm 2, and performed a first experiment with them in the state graph constructed in Section IV-A. We chose a start and goal state that are far away from each other, and we let nine cylinder-shaped obstacles move around in the scene such that they heavily impede the robot (see Fig. 1). For the robot, we used a VRML-model consisting of 11,960 triangles. The experiments were performed on an Intel Core2-6400, 2.13GHz with 2GByte of memory, and we used Solid as collision-checker [3]. Calculating the distances of the states in the state graph to the goal state to provide heuristics for the search took 0.03 seconds. Finding a path avoiding the moving obstacles with Algorithm 2 took 1.51 seconds, of which 1.31 seconds were spent on 5,844 collision-checks. With Algorithm 1, it took 4.80 seconds, of which 4.67 seconds were spent on 21,373 collision-checks.

Thus, Algorithm 2 saves almost a factor 4 in the number of collision-checks, at the cost of a slightly increased combinatorial overhead. The collision-checks are the major factor in the running time of both algorithms, but we note that the relative share of the collision-checks in the total running time depends on how costly the collision-checks are, which in turn depends on the complexity of the objects used to model the robot and the obstacles. We used a fairly complicated robot, but simple obstacles, which we believe results in collision-checks of costs typical for many applications. Note that if we would not exploit the cylindrical structure of the obstacles in the state-time space, the difference in the number of collision-checks between the two algorithms is much higher:

TABLE I

RESULTS FOR VARIOUS VALUES OF Δt

		Algorithm 2			Algorithm 1		
Δt	#states	#col	col-t	tot-t	#col	col-t	tot-t
0.10	576,769	59,840	11.80	16.88	211,027	41.13	44.00
0.15	164,793	9,257	2.01	2.42	39,095	8.05	8.34
0.20	75,609	5,844	1.31	1.51	21,373	4.67	4.80
0.25	37,245	3,739	0.88	1.00	14,966	3.49	3.57

8,976 for Algorithm 2 and 63,193 for Algorithm 1.

We also varied the value of Δt, the parameter determining the resolution of the discretization. Note that the sizes of state graph and the state-time graph grow quickly when Δt gets smaller, as $\Delta v \sim \Delta t$ and $\Delta s \sim \Delta t^2$. Table I gives the results for different values of Δt in the same setting as the above experiment. The results give an indication how the performance of the algorithms will scale with problems of different sizes. The columns named "#col" indicate the number of collision-checks used, "col-t" the amount of time spent on collision-checks (in seconds), and "tot-t" the total time needed to plan a path (in seconds).

As can be seen from the table, both algorithms scale similarly as the size of the state graph increases. Algorithm 1 uses a factor 4 more collision-checks than Algorithm 2, and for both algorithms, the relative amount of time spent on collision-checks remains more or less the same.

VI. CONCLUSION AND FUTURE WORK

In this paper, we have presented a new method for kinodynamic planning in dynamic environments. Our method extends the method of [6] to work on a roadmap instead of a path, thereby greatly expanding the maneuverability of the robot. It also extends the method of [1], so that it takes the kinematic and dynamic constraints on the robot into account. An advantage of our method with respect to [8] is that narrow passage problems caused by the static obstacles are tackled already during preprocessing (by building a roadmap), so that they do not affect the query performance.

We implemented our method, and experimental results show that natural and realistic paths can be computed in complicated dynamic environments in the order of a second of running time using a discretization of a reasonably high resolution. This makes the algorithm perfectly suited for many applications, for instance prioritized multi-robot motion planning [2].

Our paper contains other contributions. Firstly, we have shown how roadmaps can be created for car-like robots containing visually attractive paths with a continuous curvature profile. Also, we have exploited the specific nature of the graphs in state-time space, and presented a search algorithm that is faster than the traditional A* approach. It works particularly well for graphs of the type we have seen in our paper, but it is applicable to any kind of implicit graph or grid where collision-checks or other expensive operations have to be carried out during the search to check the traversability of states. Examples are coordinated multi-robot planning [12] and Lazy PRM [4].

An interesting direction for future work is to make the method suitable for *real-time application*. In real-time settings, one is given a planning query from state q_{start} to q_{goal} at some real-world time t_w, which should be answered as quickly as possible. Thus, we should reserve some amount of time τ for planning, and initialize the planner with start state-time $\langle q_{\mathrm{start}}, t_w+\tau \rangle$. The planner *must* finish within time τ, otherwise the plan is invalid. However, for our method, as well as the previously proposed approaches [1], [6], [8], the amount of time needed to plan a path is difficult to estimate accurately in advance. Hence, it is difficult to choose an appropriate value for τ. Moreover, it is impossible to guarantee that it finishes within this time. A solution to this problem is to have a planner that is given τ time to initialize, and then computes a path simultaneously with its execution. Our planner is easily adapted to this situation, as it can use the last considered path in the algorithm as an indication of the global direction towards the goal. Details need to be worked out, but this remains a subject of future research.

REFERENCES

[1] J. van den Berg, M. Overmars. Roadmap-based motion planning in dynamic environments. *IEEE Trans. on Robotics* 21(5), pp. 885–897, 2005.

[2] J. van den Berg, M. Overmars. Prioritized motion planning for multiple robots. *Proc. IEEE/RSJ Int. Conf. on Intelligent Robots and Systems*, pp. 2217–2222, 2005.

[3] G. van den Bergen. *Collision detection in interactive 3D environments*. Morgan Kaufmann Publishers, San Francisco, 2004.

[4] R. Bohlin, L. Kavraki. Path planning using Lazy PRM. *Proc. IEEE Int. Conf. on Robotics and Automation*, pp. 521–528, 2000.

[5] B. Donald, P. Xavier, J. Canny, J. Reif. Kinodynamic planning. *Journal of the ACM* 40, pp. 1048–1066, 1993.

[6] T. Fraichard. Trajectory planning in a dynamic workspace: a 'state-time space' approach. *Advanced Robotics* 13(1), pp. 75–94, 1999.

[7] R. Geraerts, M. Overmars. Creating high-quality roadmaps for motion planning in virtual environments. *Proc. IEEE/RSJ Int. Conf. on Intelligent Robots and Systems*, pp. 4355–4361, 2006.

[8] D. Hsu, R. Kindel, J.-C. Latombe, S. Rock. Randomized kinodynamic motion planning with moving obstacles. *Int. Journal of Robotics Research* 21(3), pp. 233–255, 2002.

[9] L. Kavraki, P. Švestka, J.-C. Latombe, M. Overmars. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *IEEE Trans. on Robotics and Automation* 12(4), pp. 566–580, 1996.

[10] S. Koenig, Y. Smirnov. Sensor-based planning with the freespace assumption. *Proc. IEEE Int. Conf. on Robotics and Automation*, pp. 3540–3545, 1997.

[11] S. Koenig, M. Likhachev, D. Furcy. Lifelong planning A*. *Artificial Intelligence* 155, pp. 93–146, 2004.

[12] S. LaValle, S. Hutchinson. Optimal motion planning for multiple robots having independent goals. *IEEE Trans. on Robotics and Automation* 14(6), pp. 912–925, 1998.

[13] S. LaValle, J. Kuffner. Randomized kinodynamic planning. *Int. Journal of Robotics Research* 20(5), pp. 378–400, 2001.

[14] S. LaValle. *Planning Algorithms*. Cambridge University Press, New York, 2006.

[15] D. Nieuwenhuisen, A. Kamphuis, M. Mooijekind, M. Overmars. Automatic construction of roadmaps for path planning in games. *Proc. Int. Conf. on Computer Games, Artificial Intelligence, Design and Education*, pp. 285–292, 2004.

[16] J. Pettré, T. Siméon, J.P. Laumond. Planning human walk in virtual environments. *Proc. IEEE/RSJ Int. Conf. on Intelligent Robots and Systems*, pp. 3048–3053, 2002.

[17] S. Petty, T. Fraichard. Safe motion planning in dynamic environments. *Proc. IEEE/RSJ Int. Conf. on Intelligent Robots and Systems*, pp. 3726–3731, 2005.

[18] A. Scheuer, C. Laugier. Planning sub-optimal and continuous curvature paths for car-like robots. *Proc. IEEE/RSJ Int. Conf. on Intelligent Robots and Systems*, pp. 25–31, 1998.