Dynamic Path Planning for Coordinated Motion of Multiple Mobile Robots

Marco Langerwisch and Bernardo Wagner

Abstract—This paper presents an approach of calculating coordinated paths for multiple mobile robots operating in the same environment. Therefore, the presented approach extends the configuration space with a time component. Based on an A* algorithm, it calculates collision free paths of multiple robots. Moreover, it is capable of replanning the path in dynamic environments without doing an iterated A* path planning from scratch. The correctness of the approach is proven, and experiments show the achieved improvement in computation time compared to an iterated path planning from scratch. A lattice type state space representation based on short motion sequences is applied to the approach, resulting in more realistic and traversable trajectories for vehicles. The feasibility is shown by letting a simulated convoy turn autonomously to the opposite direction.

I. INTRODUCTION

The avoidance of collisions of multiple mobile robots operating in the same environment is a crucial task in robot navigation. Either the robots can use local obstacle avoidance mechanisms, or their paths have to be coordinated. The motivation for this contribution is a coordination in path planning of multiple autonomous robot systems that is able to calculate collision free paths in advance, and is able to react on dynamic changes in the environment.

Common approaches of path planning for single autonomous robots are based on grid-based obstacle representation [1], or use potential fields to model obstacles in the environment [2]. We restrict ourselves to grid-based maps, built by incorporated laser rangefinder measurements. The most popular approach in the field of path determination given start and goal points and a map is the graph-based A* algorithm [3]. It extends Dijkstra’s shortest path algorithm with a heuristic to focus the search in direction of the goal position.

In reality, a complete map of the environment is not available for path planning. Hence, a trajectory for the robots movement is calculated out of a local map. Advancing to the goal point, the robot detects unknown obstacles or even unexpected free space by its onboard sensors. Using A*, one has to do a full path search at each update step, resulting in a possibly very complex computation at a time. Koenig and Likhachev [4] extended the basic A* algorithm to the Lifelong Planning A* (LPA*) algorithm, incorporating incremental changes in the environment over time. They reuse previous information to update the path, without recomputing a full A* search, resulting in a great improvement in time requirement. In [5] and [6], Stentz also extended the basic A* algorithm for the dynamic navigation of moving robots, using the same principle as the LPA*, called the Dynamic A* (D*) algorithm. A more efficient version of D* has been given in [7], called D* Lite.

In case of multiple mobile robots operating in the same environment, two classes of approaches for path coordination can be identified. Approaches with central coordination treat the single robots as one composite robot, resulting in a unified state space. Standard path planning algorithms like graph-based approaches can be applied. Unfortunately, the dimension of the state space is usually very large, resulting in a high time complexity of the computation. An attractive alternative is the class of decoupled planning approaches.

Either, each robot plans its own path and afterwards they are somehow coordinated [8]. Or, these approaches use priority schemes where robots with lower priorities have to incorporate the movements of higher prioritized robots [9]. The advantage is the low time complexity of each single planning step. The drawback is the lack of optimality due to the predefined priorities. In [10], an approach is presented to overcome this drawback by a dynamic prioritization.

In this contribution, we will apply the approach of extending the configuration space of each robot with the time dimension, called C-Space-time [11]. We will use a simple priority scheme for decoupled path planning. Moreover, we will extend our algorithm to cope with computations in C-Space-time and handle dynamic environments as D*, maintaining path optimality.

Path calculation based on C-Space-time has to plan a path in forward direction due to the subsequent timing steps. Because D* and D* Lite are based on backward path calculation, they can not directly be applied to our purposes. Based on A* forward computation, our approach is capable of incorporating higher prioritized paths and plan a new path in dynamic environments. We will proof correctness and optimality of the resulting trajectories, and show the application to a multi-robot scenario.

Moreover, a path is usually calculated based on grid coordinates, resulting in trajectories that are not drivable by non-holonomic mobile robots. Therefore, we will extend our approach to calculate a trajectory consisting of short kinematically feasible motion sequences, similar to the calculation of a lattice type state space [12].

The paper is organized as follows. The subsequent section contains a motivating example of multiple robots moving in the same environment. Section III repeats the basics of
the LPA* algorithm. The application to C-Space-time and the use of feasible motion sequences is described in section IV. Experimental results will be presented in section V. The paper ends with a conclusion and an outlook on future work.

II. MOTIVATING EXAMPLE

Let us look at the situation depicted in Fig. 1a. We have an eight-connected grid with obstacle grid cells marked in black. Two holonomic robots with point dimensions are placed in the environment. The start positions of two robots are marked with S1 and S2, the goal positions with G1 and G2. The grid cell of the current robot position is filled in light blue (robot 1) and light green (robot 2), respectively. We assume that the higher prioritized robot 1 is capable of dynamic replanning, robot 2 has to have the same capabilities. One approach would be to do a full LPA* replanning, resulting in full replanning for robot 2. To avoid this full replanning, our contribution will extend the LPA* to deal with dynamic environments in the space and time dimensions.

Please note again, that D* and D* Lite cannot be applied here due to their backward search (goal to start). Forward search is necessary to calculate the needed time for movements, especially when the needed time depends on more factors than on the number of grid cells visited.

III. LIFELONG PLANNING A*

Lifelong Planning A* (LPA*) is a graph-based algorithm for calculating an optimal path, i.e. a path with minimal costs, between a start vertex $s_{start}$ and a goal vertex $s_{goal}$. $S$ is the set of all vertices in the graph, hence $s_{start} \in S$ and $s_{goal} \in S$. Moreover, it incorporates changes in the edge costs incrementally, and updates the calculated path accordingly. LPA* can be used for grid-based path planning, using the occupancy values as cost function, and the goal distance as a heuristic to find the search in direction of the goal vertex.

The full algorithm is depicted in Alg. 1. $Succ(s) \subseteq S$ and $Pred(s) \subseteq S$ denote the set of successors of vertex $s$, and the set of predecessors, respectively. The cost of moving from vertex $s$ to vertex $s'$ is denoted as $0 \leq c(s, s') \leq \infty$. $h(s)$ is the heuristic and satisfies $h(s_{goal}) = 0$ and $h(s) \leq c(s, s') + h(s')$ for all vertices $s \in S$ and $s' \in Succ(s)$ with $s \neq s_{goal}$. The $g$-value of each vertex is the start distance, i.e. the accumulated costs travelling from the start vertex to the current vertex. The $rhs$-value of a vertex is a one-step lookahead value based on the $g$-values. A vertex is called locally consistent if $g$- and $rhs$-values of the vertex are equal. If $g(s) < rhs(s)$, the vertex $s$ is called locally underconsistent, and locally overconsistent if $g(s) > rhs(s)$. $U$ is a priority queue and contains all inconsistent vertices. $U$ is ordered by a key $k(s)$ for all vertices $s \in S$. It consists of two components: $k(s) = [k1(s); k2(s)]$, where $k1$ and $k2$ are calculated according to line 2 of Alg. 1. Some functions are used to manage the priority queue. U.TOP Key() returns the smallest priority of all vertices in $U$, and $[\infty; \infty]$ if $U$ is empty. U.Pop() deletes the vertex with the smallest priority in the priority queue and returns the vertex. U.Insert(s, k) inserts vertex s into $U$ with priority $k$. U.Remove(s) removes vertex s from $U$. The execution of lines 25-36 is referred to as vertex expansion.

The algorithm first initializes the $g$- and $rhs$-values of

<table>
<thead>
<tr>
<th>Algorithm 1 Lifelong Planning A*</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: procedure CALCULATE Key(s)</td>
</tr>
<tr>
<td>2: return $[\min(g(s), rhs(s)) + h(s); \min(g(s), rhs(s))]$</td>
</tr>
<tr>
<td>3: end procedure</td>
</tr>
<tr>
<td>4: procedure INITIALIZE( )</td>
</tr>
<tr>
<td>5: $U := \emptyset$</td>
</tr>
<tr>
<td>6: for all $s \in S$ do</td>
</tr>
<tr>
<td>7: $rhs(s) \leftarrow g(s) \leftarrow \infty$</td>
</tr>
<tr>
<td>8: end for</td>
</tr>
<tr>
<td>9: U.Insert($s_{start}$, $h(s_{start})$); 0)</td>
</tr>
<tr>
<td>10: end procedure</td>
</tr>
<tr>
<td>11: procedure UPDATE Vertex(u)</td>
</tr>
<tr>
<td>12: if $u \neq s_{start}$ then</td>
</tr>
<tr>
<td>13: $rhs(u) \leftarrow \min_{s' \in Predecessor(u)}(g(s') + c(s', u))$</td>
</tr>
<tr>
<td>14: end if</td>
</tr>
<tr>
<td>15: if $u \notin U$ then</td>
</tr>
<tr>
<td>16: U.Insert(u, CALCULATE Key(u))</td>
</tr>
<tr>
<td>17: end if</td>
</tr>
<tr>
<td>18: if $g(u) \neq rhs(u)$ then</td>
</tr>
<tr>
<td>19: U.Remove(u)</td>
</tr>
<tr>
<td>20: end if</td>
</tr>
<tr>
<td>21: end procedure</td>
</tr>
<tr>
<td>22: procedure COMPUTE Shortest Path( )</td>
</tr>
<tr>
<td>23: while U. Top Key() &lt; CALCULATE Key($s_{goal}$)</td>
</tr>
<tr>
<td>24: OR $rhs(s_{goal}) \neq g(s_{goal})$ do</td>
</tr>
<tr>
<td>25: $u = U.Pop()$</td>
</tr>
<tr>
<td>26: if $g(u) &gt; rhs(u)$ then</td>
</tr>
<tr>
<td>27: $g(u) \leftarrow rhs(u)$</td>
</tr>
<tr>
<td>28: for all $s \in Succ(u)$ do</td>
</tr>
<tr>
<td>29: UPDATE Vertex(s)</td>
</tr>
<tr>
<td>30: end for</td>
</tr>
<tr>
<td>31: else</td>
</tr>
<tr>
<td>32: $g(u) \leftarrow \infty$</td>
</tr>
<tr>
<td>33: for all $s \in Succ(u) \cup {u}$ do</td>
</tr>
<tr>
<td>34: UPDATE Vertex(s)</td>
</tr>
<tr>
<td>35: end for</td>
</tr>
<tr>
<td>36: end if</td>
</tr>
<tr>
<td>37: end while</td>
</tr>
<tr>
<td>38: end procedure</td>
</tr>
<tr>
<td>39: procedure MAIN( )</td>
</tr>
<tr>
<td>40: Initialize()</td>
</tr>
<tr>
<td>41: loop</td>
</tr>
<tr>
<td>42: COMPUTE Shortest Path( )</td>
</tr>
<tr>
<td>43: Wait for changes in edge costs</td>
</tr>
<tr>
<td>44: for all directed edges $(u, v)$ with changed costs do</td>
</tr>
<tr>
<td>45: Update the edge cost $c(u, v)$</td>
</tr>
<tr>
<td>46: UPDATE Vertex(s)</td>
</tr>
<tr>
<td>47: end for</td>
</tr>
<tr>
<td>48: end loop</td>
</tr>
<tr>
<td>49: end procedure</td>
</tr>
</tbody>
</table>
all vertices with $\infty$, and the priority queue $\mathcal{U}$ with the start vertex $s_{start}$. To find a shortest path, lines 25-36 are executed as long as the goal vertex is inconsistent or there is a chance to find a shorter path to the goal vertex. Lines 27-30 expand overconsistent vertices, and 32-35 expand underconsistent vertices. If \textsc{ComputeShortestPath()} terminates and $g(s_{goal}) \neq \infty$, a path with finite costs from the start vertex to the goal vertex has been found. To get it, one has to trace back from $s_{goal}$ to $s_{start}$, always moving from vertex $s$ to its predecessor $s'$ minimizing $g(s') + c(s', s)$.

After calculation of a path, LPA* waits for changed edge costs in line 43. Affected vertices are updated and hence their $\text{rhs}$-values and keys are recalculated. Their membership to the priority queue is updated accordingly. Finally, a new shortest path is calculated.

Some optimizations can be applied to the algorithm described above. For details and a detailed description of the LPA* algorithm please refer to [4] and [13].

IV. TIME D*

The idea behind our contribution is to add the time component to path calculation, and let the algorithm react on dynamic environments while advancing on its trajectory. Our approach will be called \textit{Time D*} (TD*) and is based on LPA*. All line numbers in this section refer to Alg. 2. The following subsection introduces the extension of LPA* to C-Space-time, and the added motion of the robot. Section IV-B describes the handling of dynamic environments, and in section IV-C we will proof the correctness of the algorithm and have a look at the worst case time requirements. The last subsection presents the application of a lattice type state space to the algorithm.

A. Adding Time Dimension and Motion

To extend the original LPA* algorithm to the time dimension, called C-Space-time [11], the configuration space and the map are simply extended by a time component. This results in time-dependent costs for moving from one vertex to another. Hence, the definition of the cost function changes to $0 \leq c(s, s', t) \leq \infty$, having $s \in S$, $s' \in \text{Succ}(s)$, and $t$ the point in time of the movement from $s$ to $s'$. $t$ could be a discrete time step or a certain period of time, depending on the implementation. We use intervals, similar to safe and collision intervals introduced in [14]. Due to the necessity of knowing the exact point in time when calculating the costs for traversing from one vertex to another, we have to start our search at vertex $s_{start}$ and expand the search space towards $s_{goal}$. This is contrary to the D* [6] and D* Lite [7] approaches and results in some drawbacks compared to these approaches, as described later.

The motion of the robot is easily added to the algorithm, similar to [7]. In line 49, the next vertex on the robot path is assigned to $s_{start}$. Due to the forward search used here, one has to trace back from $s_{goal}$ to $s_{start}$, and take the previous vertex as the new start vertex. One could either trace back as described in section III using the $g$- and $c$-values, or use backpointers as in [6], e.g. Execution of line 50 moves the robot to the designated vertex.

B. Dynamic Environment

A moving robot is acquiring new sensor information while moving on its trajectory, resulting in a map update at each time step. Moreover, if using a multi-robot system, the trajectories of higher prioritized robots might have changed and have to be treated as dynamic obstacles in the map. One approach would be to do a full A* or LPA* path planning from scratch. In many cases, previous calculated information can be reused for updating the path, resulting in a less complex computation compared to a path planning from scratch. Furthermore, most of the changes in the environment do not affect the trajectory of the robot and hence should not result in a recomputation of the planned path.

The LPA* algorithm is capable of handling dynamic environments on a static robot. When planning a path for
Algorithm 2 Time D*

1: procedure CALCULATE KEY(s)
2: return \([\min(g(s), rhs(s)) + h(s); \min(g(s), rhs(s))]\)
3: end procedure

4: procedure INITIALIZE( )
5: \(U := \emptyset\)
6: for all \(s \in S\) do
7: \(rhs(s) \leftarrow g(s) \leftarrow \infty\)
8: end for
9: U.INSERT(sstart, [h(sstart); 0])
10: end procedure

11: procedure UPDATEVERTEX(u)
12: if \(u \neq s_{start}\) then
13: \(rhs(u) \leftarrow \min_{s \in Pred(u)}(g(s') + c(s', u))\)
14: end if
15: if \(u \in U\) then
16: U.REMOVE(u)
17: end if
18: if \(g(u) \neq rhs(u)\) then
19: U.INSERT(u, CALCULATE KEY(u))
20: end if
21: end procedure

22: procedure COMPUTESHORTTESTPATH( )
23: while U.TOPKEY( ) < CALCULATE KEY(sgoal) do
24: \(u = U.TOP(\)\)
25: if \(g(u) > rhs(u)\) then
26: \(g(u) \leftarrow rhs(u)\)
27: for all \(s \in Succ(u)\) do
28: UPDATEVERTEX(s)
29: end for
30: else
31: \(g(u) \leftarrow \infty\)
32: for all \(s \in Succ(u) \cup \{u\}\) do
33: UPDATEVERTEX(s)
34: end for
35: end if
36: end while
37: end procedure

39: procedure MAIN( )
40: INITIALIZE( )
41: COMPUTESHORTTESTPATH( )
42: while \(s_{start} \neq s_{goal}\) do
43: if \(g(s_{goal}) = \infty\) then
44: exit
45: end if
46: INITIALIZE( )
47: COMPUTESHORTTESTPATH( )
48: else
49: \(s_{start} \leftarrow \) next vertex on path
50: Move robot to \(s_{start}\)
51: Scan graph for changes in edge costs
52: if any edge cost changed then
53: for all directed edges \((u, v)\) with changed costs do
54: Update edge cost \(c(u, v)\)
55: end for
56: COMPUTESHORTTESTPATH( )
57: end if
58: end while
59: end procedure

C. Theoretical Analysis

In this section, we will proof the most important property of the algorithm, the correctness. Finally, we will have a look at the worst case time requirement.

**Theorem 1:** The Time D* algorithm always finds a shortest path from \(s_{start}\) to \(s_{goal}\), if at least one path exists.

**Proof:** Two cases have to be considered, if a path from \(s_{start}\) to \(s_{goal}\) has not been found (lines 46-47), i.e. is invalid, and if a path is replanned (lines 49-58) and valid. The proof for the first case directly follows from the correctness of the LPA* algorithm [13], because a path is planned from scratch. If, in the second case, the start vertex changes from \(s_{startOld}\) to \(s_{start}\) (line 49), a shortest path has been found from \(s_{startOld}\) to \(s_{goal}\), which is correct (proof in [13]). If \(s_{start}\) is on the optimal path, i.e. \(s_{start} \in Succ(s_{startOld})\) and \(s_{start}\) can be reached when tracing back from \(s_{goal}\), always moving from the current vertex \(s\) to any \(s' \in Pred(s)\), having \(g(s) = g(s') + c(s', s, t)\), the path from \(s_{start}\) to \(s_{goal}\) is also a shortest path. If no such path can be found, the path is considered as invalid and replanned from scratch. Correctness is as shown in the first case.

**Theorem 2:** If a locally overconsistent vertex is selected for expansion on line 25, then it is locally consistent the next time lines 23-24 are executed and remains locally consistent until COMPUTESHORTTESTPATH( ) terminates.

**Theorem 3:** COMPUTESHORTTESTPATH( ) expands a vertex at most twice, namely at most once when it is locally underconsistent and at most once when it is locally overconsistent, and thus terminates.

Proofs for Theorems 2 and 3 can be found in [13].

Looking at the worst case time requirement, different cases have to be considered. First, in a path planning from scratch, i.e. a call to INITIALIZE( ) followed by a call to COMPUTESHORTTESTPATH( ), all vertices except \(s_{start}\) are locally consistent with \(g\)- and \(rhs\)-values of \(\infty\). While expanding vertices from the priority list \(U\), some vertices become locally underconsistent and at most once when it is locally overconsistent, and thus terminates.
expanded at most twice in this case. If a replanning results in an invalid path, a path planning from scratch is initialized. Hence, a vertex can be expanded at most three times during replanning. This is a drawback compared to D* and D* Lite, where a vertex is expanded at most twice [7], but we will show that an improvement compared to repeated full path planning from scratch is achieved.

D. Lattice Type State Space

Usually, paths planned in grid-based maps either assume an holonomic robot, or a smoothing of the calculated path in a post-processing step, due to sharp turns that result from the grid-based representation. To make the path planning more realistic, we apply a Lattice type state space to calculate with short motion primitives [12] (motion segments) instead of lines connecting the centres of grid cells. These motions primitives are either curves with constant radius \( r \) or straight lines \( (r = 0) \), each with a predefined length \( l \). A minimum and a maximum turning angle of the vehicle are defined as \( \omega_{\min} \) and \( \omega_{\max} \), respectively.

The state space is calculated online, i.e. during path planning. Each vertex of the state space consists of a feasible motion segment and its end point in cartesian coordinates. Instead of expanding the eight neighbouring nodes, as in case of an eight-connected grid, for each vertex, \( n \) feasible motion segments in forward, and \( n \) feasible motion segments in backward direction are expanded, starting at the current vertex and resulting in \( q = 2 \cdot n \) new vertices. \( n \) has to be odd to include the straight lines in both directions. Because the vertices are calculated at runtime, the set of all vertices \( S \) is not known in advance. Hence, \( s_{\text{goal}} \) has to be defined differently. \( g(s_{\text{goal}}) \) and \( rhs(s_{\text{goal}}) \) are initialized with \( \infty \). Each time the endpoint of a vertex \( s \) is in \( \epsilon \)-distance to \( p_{\text{goal}} \), the vertex is a candidate to be \( s_{\text{goal}} \). It becomes \( s_{\text{goal}} \), if \( k(s) < k(s_{\text{goal}}) \). To get the cost \( c(s, s', t) \) of traversing along a directed edge between two vertices \( s \) and \( s' \), the vehicle’s footprint is projected into the grid map, and the maximum occupancy value of all covered grid cells is taken as obstacle cost. Moreover, factors as the turning radius or the move direction are taken into account for calculation of \( c(s, s', t) \), and the velocity of the motion segment and hence the time required to traverse it depend on the turning radius \( r \).

The application of the lattice type TD* algorithm is shown in section V-B.

V. EXPERIMENTAL RESULTS

Both variants of the Time D* algorithm were tested in simulation. The following subsection presents the results of a series of experiments with grid-based path planning and shows the achieved improvement compared to an iterated path planning from scratch. An experiment in section V-B demonstrates the feasibility of multi-robot path planning based on short motion segments.

A. Grid-Based Path Planning

To show the improvement of TD* compared to full path planning from scratch, we conducted a series of MATLAB-simulations with TD* and an LPA* implementation, both in their basic versions without further optimizations. Each time when TD* moves the robot on its path, a path planning from scratch with LPA* was computed for comparison, called LPA* Brute-Force (LPA* BF). The scenario was an eight-connected grid map with 10x10, 20x20, and 40x40 cells, respectively. The start vertex was located at \( (2, 2) \), and the goal vertex was located at \( (9, 9) \), \( (19, 19) \), and \( (39, 39) \), respectively. At the beginning, 40% of the grid cells were randomly chosen to be occupied. After each movement of the robot, 0.5% of the free grid cells changed their state to occupied, and another 0.5% were turned into free cells. Each series consisted of 100 test runs. For each run, the total number of vertex expansions was counted, and the duration of the computation for the path planning (including all replanning steps) was summed up. For the number of vertex expansions and the duration of each test run, a speed up factor was calculated with \( \frac{\text{LPA* BF}}{\text{TD*}} \). The mean values of these factors together with their 95% confidence intervals are listed in Table I. The leftmost column describes the size of the map.

The results show that Time D* is able to find a shortest path in dynamic environments more than two times faster than doing an iterated path planning from scratch when applied to a mobile robot. The number of vertex expansions is also more than two times lower with our new approach.

B. Lattice Type Path Planning

To demonstrate the applicability of the lattice type TD*, we simulated a simple scenario where a convoy consisting of three vehicles turns autonomously. It is assumed that the vehicles in the convoy are ordered, i.e. the second vehicle follows the first, and the third vehicle follows the second. After turning the convoy, the vehicle order in direction of motion should not be changed, hence the first (1) and the last vehicle (3) need to switch locations, while the middle vehicle (2) has to turn on the spot. To avoid collisions, all paths have to be coordinated. The initial configuration of the vehicles is depicted in Fig. 3a. All three vehicles have come to a stop. The goal position of the first vehicle is the start position of the last vehicle, turned by 180°, and vice versa. Its own start position turned by 180° is the goal position of the middle vehicle. Paths planned without any coordination are depicted in Fig. 3a. One can see the collisions resulting from the trajectories. Hence, a coordinated prioritized path planning is initiated. Higher prioritized paths are incorporated into the map before line 51 in Alg. 2. Tests have shown that the best priority orders are 2, 1, 3 or 2, 3, 1 in this situation, the first was chosen here. The planned coordinated paths are depicted in Fig. 3b.

<table>
<thead>
<tr>
<th>Map size</th>
<th>Vertex expansions</th>
<th>Duration</th>
</tr>
</thead>
<tbody>
<tr>
<td>10x10</td>
<td>2.54 ± 0.20</td>
<td>2.57 ± 0.18</td>
</tr>
<tr>
<td>20x20</td>
<td>2.69 ± 0.13</td>
<td>2.65 ± 0.13</td>
</tr>
<tr>
<td>40x40</td>
<td>2.42 ± 0.11</td>
<td>2.20 ± 0.10</td>
</tr>
</tbody>
</table>
After some time period, the environment changes as a new obstacle, e.g. a pedestrian, occurs. All three vehicles calculate a new path, each incorporating the higher prioritized path(s). The obstacle and the replanned paths are depicted in Fig. 3c. All vehicles can proceed advancing towards their goal points. At the end, the convoy has turned (see Fig. 3d), and could resume its travel. Note, that the vehicles are not exactly located at their goal points due to the ε-criterion (see section IV-D).

The experiment shows the application of a more realistic way of state space representation to a graph-based path planning algorithm. It is shown that the calculated paths are correct and can be replanned during operation. Drawbacks of this approach are the high requirements for memory and computation time due to the online calculation of the motion segments (contrary to predefined grid nodes). Hence, currently the dynamic replanning of the lattice type TD* algorithm is not computable in real time. Offline precalculation of the state space or a multi-resolution lattice [12] may be applied. This will be part of future work.

VI. CONCLUSION

This paper introduced a new approach to plan paths for mobile robots in changing environments when a coordination between the robots is wanted to avoid collisions. Therefore, the Lifelong Planning A* (LPA*) algorithm has been extended to cope with the time dimension and moving robots. The resulting algorithm, called Time D*, is capable of planning a path based on a time-annotated map containing dynamic obstacles. Moreover, it can replan its path when changes in the environment occur more than two times faster than an iterated LPA* algorithm from scratch. A path representation based on short motion sequences has been applied to the new approach, resulting in more realistic and traversable trajectories for vehicles. The feasibility has been shown by letting a simulated convoy turn autonomously to the opposite direction.

Future work will deal with the further reduction of necessary path plannings from scratch, and the real time replanning of feasible paths.

REFERENCES