A Viterbi-like Approach for Trajectory Planning with Different Maneuvers

Jörg Roth

Department of Computer Science, Nuremberg Institute of Technology, Kesslerplatz 12,
90489 Nuremberg, Germany
Joerg.Roth@th-nuernberg.de

Abstract. The task of a trajectory planning tries to find a sequence of driving commands that connects two configurations, whereas we have to consider nonholonomic constraints, obstacles and driving costs. In this paper, we present a new approach that supports arbitrary primitive trajectories, cost functions and constraints. The vehicle's driving capabilities are modeled by a list of supported maneuvers. For maneuvers there exist equations that map configurations to driving commands. From all possible maneuver sequences that connect start and target, we compute the optimum with a Viterbi-like approach.

1 Introduction

Trajectory planning is a fundamental function of a mobile robot. When executing complex tasks such as transporting goods, the robot has to drive trajectories that meet certain measures of optimality. For this, a cost function may consider driving time, energy consumption, mechanical wear or buffer distance to obstacles.

Whereas a geometric *route planning* tries to find a line string with minimal costs that does not cut an obstacle (with respect to the robot's driving width), the *trajectory planning* also considers nonholonomic constraints such as curve angles or orientations. Our approach is based on three key ideas:

- A first route planning step reduces the overall complexity of the problem, as it only considers workspace dimensions.
- We model the measure of optimality as arbitrary cost function that considers obstacles. It may use different mechanisms to check for obstacles, i.e. grids or spatial indexing to speed up execution.
- The robot's driving capabilities are modeled by a list of *primitive trajectories* (e.g. straight forward, arc, clothoid) and *maneuvers* (small sequences of primitive trajectories). These lists are also considered as black box for our algorithm. The benefit: our trajectory planning is able to support different types of vehicles without changing the overall algorithm.

Our approach is able to compute suitable trajectories in short runtime that *linearly* depends on the number of intermediate points. Moreover, the first driving command converges very fast. In practice it is available in *constant time*. This means, a mobile

robot is able to start driving very quickly with the first driving command (e.g. in emergency situations) whereas the remaining sequence is computed while driving.

The approach is successfully realized and evaluated on the *Carbot* platform.

2 Related Work

Early work investigates shortest paths for vehicles that can drive straight forward and circular curves [3, 5, 17]. Without obstacles, we can connect two configurations with only three primitive trajectories. Further work addresses the problem of discontinuities in the curvature and tries to integrate the clothoid as further primitive trajectory [2, 19]. Instead of clothoids, also cubic spirals or splines were considered [4, 12].

More related to our approach is work that investigates longer paths that go through an environment of obstacles. As the space of possible trajectory sequences gets very large, probabilistic approaches are a suitable method to find at least a suboptimal solution [7, 9, 10]. They randomly connect configurations by primitive trajectories and are able to search on the respective graph to plan an actual path. Further work uses potential fields [1] or visibility graphs [14]. With the help of geometric route planners, the overall problem of trajectory planning can be reduced. In [8], the route planning step and a local trajectory planning step are recursively applied.

Random sampling can also be used to improve generated trajectories. E.g. CHOMP [21] uses functional gradient techniques based on Hamiltonian Monte Carlo to iteratively improve the quality of an initial trajectory. The approach in [13] represented the continuous-time trajectory as a sample from a Gaussian process generated by a linear time-varying stochastic differential equation. Then gradient-based optimization technique optimizes trajectories with respect to a cost function.

Our approach was inspired by the *state lattice* idea introduced in [16]. State lattices are discrete graphs embedded into the continuous state space. Vertices represent states that reside on a hyperdimensional grid, whereas edges join states by trajectories that satisfy the robot's motion constraints. The original approach is based on equivalence classes for all trajectories that connect two states and perform inverse trajectory generation to compute the result trajectory. [6] introduced a two-step approach, with coarse planning of states based on Dynamic Programming, and a fine trajectory planning that connects the formerly generated states.

Discussion

Many prior approaches impose limitations on the set of primitive trajectories. Often, this is a result of the optimization approach or of its stochastic nature. We believe the set of primitive trajectories should be considered as black box.

Approaches that directly plan on the configuration space have to deal with many dimensions and thus large spaces for solutions. We may use probabilistic methods or discretization of configuration space parameters. However, a large environment (e.g. many rooms in a building) may already lead to an undesirable large set of possibilities, even if we ignore further configuration dimensions. Several weaknesses of ran-

dom sampling were shown in the context of probabilistic roadmaps in [11]. Due to its stochastic nature runtime can hardly determined, but this is required in critical situations. We thus selected a method that gives complete control of the scale of search space and allows to pre-determine runtime.

State lattice approaches are widely used in on-road planning. States are usually embedded into a grid of workspace dimensions. This however limits planning steps to next neighbors. The original approach allows to 'skip' multiple neighbors. However, we believe, a wider segmentation that is a result of a prior route planning is more suitable. This, e.g., would simplify the planning of long straight trajectories.

We introduce a new planning approach that is able to integrate arbitrary sets of primitive trajectories, arbitrary cost functions and is able respond in linear runtime.

3 The Trajectory Planning

3.1 Basic Considerations

We now assume the robot drives in the plane in a workspace $\mathcal{W}$ with positions (x, y). The configuration space $\mathcal{C}$ covers an additional dimension for the orientation angle, i.e. a certain configuration is defined by (x, y, θ). The goal is to find a collision-free sequence of trajectories that connects two configurations, meanwhile minimizes a cost function. To merge collisions and costs, we require the cost function to produce infinite costs for trajectories that cut obstacles.

This problem has many degrees of freedom. Whereas even small distances can be connected by an infinite number of trajectories, the problem gets worse for larger environments with many obstacles. We thus introduce the following concepts:

- A route planning that solely operates on workspace $\mathcal{W}$ computes a sequence of collision-free lines of sight (with respect to the robot's width) that minimize the costs.
- As the route planning only computes route points in $\mathcal{W}$, we have to specify additional variables in $\mathcal{C}$ (here orientation θ). From the infinite assignments, we only consider a small finite set.
- From the infinite set of trajectories between two route points, we only consider a finite set of *maneuvers*. Maneuvers are trajectories, for which we know formulas that derive the respective parameters (e.g. curve radii) from start and target configurations.
- Even though these concepts reduce the problem space to a finite set of variations, this set would by far be too large for complete checks. We thus apply a Viterbi-like approach that significantly reduces the number of checked variations to find an optimum.

We carefully separated the cost function from all planning components. We assume there is a mapping from a route or trajectory sequence to a cost value according to two rules: first, we have to assign a single, scalar value to a trajectory sequence that indicates its costs. If costs cover multiple attributes (e.g. driving time *and* battery con-

sumption), the cost function has to weight these attributes and create a single cost value. Second, a collision with obstacles has to result in infinite costs.

Cost values may take into account, e.g., the path length, the amount of turn-in-place operations, the amount of backward trajectories, changes between forward and backward driving, the amount of changes in curvature or the expected energy consumption. Also the distance to obstacles could be considered, if, e.g., we want the robot to keep a safe distance where possible.

3.2 Primitive Trajectories and Maneuvers

The basic capabilities of movement are defined by a set of *primitive trajectories*. The respective set can vary between different robots. E.g. the *Carbot* [18] is able to execute the following primitive trajectories:

- $L(\ell)$: linear (straight) driving over a distance of ℓ (may be negative for backward);
- $T(\Delta\theta)$: turn in place over $\Delta\theta$;
- $A(\ell, r)$: drive a circular arc with radius r (sign distinguishes left/right) over a distance of ℓ (that may be negative for backward);
- $C(\ell, \kappa s, \kappa t)$: clothoid over a distance of ℓ with given start and target curvatures.

We are able to map primitive trajectories directly to driving commands that are natively executed by the robot's motion subsystem. Implicitly, they specify functions $c_s \rightharpoonup c_t$ that map two configurations. For a certain configuration in $\mathcal{C}$ that defines a pose (x, y, θ), the linear trajectory, e.g., specifies a function

$$L(\ell)\colon (x, y, \theta) \rightharpoonup (x+\ell\cdot\cos(\theta), y+\ell\cdot\sin(\theta), \theta) \tag{1}$$

Due to non-holonomic constraints, it usually is not possible to reverse this mapping. I.e., for given c_s, $c_t \in \mathcal{C}$ there is in general no primitive trajectory that maps c_s to c_t.

At this point, we introduce *maneuvers*. Maneuvers are small sequences of primitive trajectories (usually 2-5 elements) that *are* able to map given c_s, $c_t \in \mathcal{C}$. More specifically:

- A maneuver is defined by a sequence of primitive trajectories (e.g. denoted *ALA* or *AA*) and further constraints. Constraints may relate or restrict the respective primitive trajectory parameters.
- For given c_s, $c_t \in \mathcal{C}$ there exist formulas that specify the parameters of the involved primitive trajectories, e.g. ℓ for L, A and C, r for A, κs, κt for C.
- Sometimes, the respective equations are underdetermined. As a result, multiple maneuvers of a certain type (sometimes an infinite number) map c_s to c_t. Thus, we need further parameters, we call *free parameters* to get a unique maneuver.

We explain the concept with the help of two maneuvers we call *S-Arcs* and *Wing-Arc* (Fig. 1). S-Arcs (Fig. 1 left) is a maneuver of type *AA*. The idea of S-Arcs is to drive two arcs with equal arc radii but with opposite left/right direction.

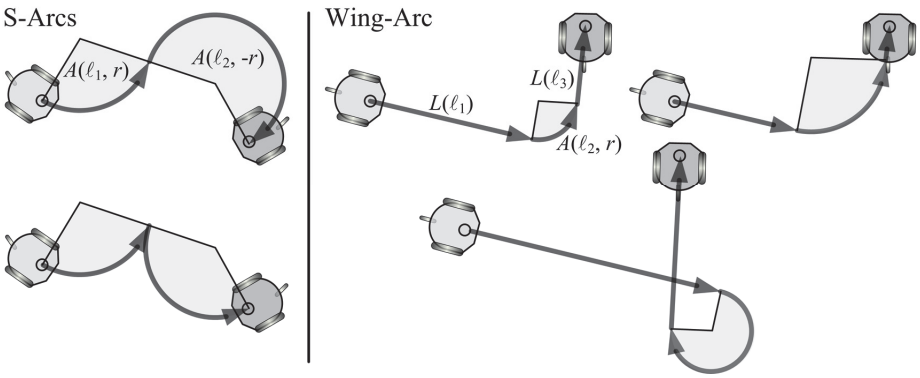

Fig. 1. Example maneuvers

Due to geometric relations, we get equations for r, ℓ_1 and ℓ_2. The calculating method: To simplify the computation, we first roto-translate start and target to move the start to (0, 0, 0) and target to (x_t', y_t', θ_t'). This leads to the formulas

$$r = \frac{y_t'(1+ct) - x_t' st - \sqrt{x_t'^2 (st^2 - 2ct + 2) + y_t'^2 (3 + ct^2) - 2x_t' y_t' st(1+ct)}}{2(ct-1)} \tag{2}$$

$$\ell_1 = r \cdot \arccos\left(\frac{(1+ct)}{2} - \frac{y_t'}{2r}\right), \; \ell_2 = \ell_1 - r \cdot \theta_t' \tag{3}$$

where st=sin(θ_t'), ct=cos(θ_t'). Actually, we get two solutions for ℓ_1, ℓ_2 parameters as arcs can always be driven in two directions (clockwise or counter-clockwise). For S-Arcs we get a total of 4 combinations. Thus, the free parameters for S-Arcs define the arc senses of rotation. It depends on the application to evaluate the respective combinations, e.g. if we want to avoid to change forward to backward or vice versa while driving.

Wing-Arc (Fig. 1 right) is a maneuver of type *LAL*. As we have a total of four parameters (r, ℓ_1, ℓ_2 and ℓ_3) we have an underdetermined set of equations, thus we have a single free parameter r (besides the arc sense of rotation).

For the Carbot project, we identified 8 maneuvers (Table 1). We assigned names that illustrate the maneuver's shape, e.g. the J-Arc drives a path that looks like the letter 'J'. The Dubins-Arcs correspond to the combination with three arcs of Dubins original approach [5]. From the maneuvers above, ∫-Arcs can be considered as a 'Swiss knife': it allows reaching any target configuration without a turn in place whereas the middle linear trajectory spans a reasonable distance to the target.

We are able to replace a single arc by two clothoids that are connected with the same curvature and end with curvature zero [15]. With these maneuvers, trajectories can be planned that did not have to deal with discontinuous curvatures.

Let Π denote the set of maneuver types relevant for a certain scenario or application. Note that Π can easily be extended or reduced. E.g. a certain application could avoid all maneuvers that contain a *T*.

Table 1. Available maneuvers in the Carbot project

Maneuver	Pattern	Free Parameters
1-Turn	*LTL*	*no*
2-Turns	*TLT*	*no*
J-Bow	*LA, LCC*	single arc sense
J-Bow2	*AL, CCL*	single arc sense
∫-Arcs	*ALA, CCLCC*	two arc radii and senses
S-Arcs	*AA, CCCC*	two arc senses
Wing-Arc	*LAL, LCCL*	single arc radius
Dubins-Arcs	*AAA, CCCCCC*	arc radius for all three arcs

We also may invent new maneuvers to increase the overall driving capabilities. For a new maneuver, we only have to set up equations that assign the respective trajectory parameters from start and target configuration.

3.3 Finding an Optimal Variation

Let $p_1=(x_1, y_1)\ldots p_n=(x_n, y_n)$ denote a route found by the route planning component for a start (p_1, θ_1) and target (p_n, θ_n). Our problem is to find a sequence of maneuvers (and thus primitive trajectories) that connects start, target and all route points in-between. We have to face two problems:

- Apart from θ_1, θ_n the orientation angles are not specified. If our route has more than two route points, we thus have an infinite set of possible intermediate orientations.
- Some maneuver types have free parameters with an infinite set of possible assignments, e.g. r for Wing-Arc.

Of the infinite number of intermediate orientations and maneuver parameters we define a finite set of promising candidates. This obviously leads to sub-optimal results. However, in reality, it does not significantly affect the overall trajectory costs. Let O_i denote all orientation candidates for a route point $i\geq 2$. In our experiments $|O_i|=5$; it contains variations of angles from the previous and to the next route point.

For free parameters we distinguish the small set of variations for arc senses (e.g. for J-Bow) and the infinite set for arc radii (e.g. for Wing-Arc). For the first type we are able to iterate through all variations. For the second type, we select a small set of candidates, similar to orientation angles. This set could be $\{r_{min}, 3\cdot r_{min}, 5\cdot r_{min}\}$, where r_{min} is the minimal curve radius. In our algorithm, $params(M)$ denotes the set of parameters for a maneuver type $M\in\Pi$. For maneuver types with no free parameters (e.g. 1-Turn), we define $params(M)=\{\varnothing\}$, whereas $\varnothing$ is an empty parameter setting.

Even though we now 'only' have a finite set of variations $\Pi\times O_i\times params(M)$ for a single route step, the number of variations still is by far too large for a complete check. To give an impression: for 5 route points we get a total number of 20 million, for 20 route points $2\cdot 10^{37}$ permutations (for an average of 20 maneuvers and 5 inter-

mediate angles). Obviously, we need an approach that computes an appropriate result without iterating through all permutations.

Our approach is inspired by the Viterbi algorithm [20] that tries to find the most likely path through hidden states. To make use of this approach, we replace 'most likely' by 'least costs', and 'hidden states' by 'unknown parameters'. We thus look for a sequence of maneuvers/orientations/free parameters that connect them with minimal costs.

This approach is suitable, because optimal paths have a characteristic: the interference between two primitive trajectories in that path depends on their distance. If they are close, a change of one usually also causes a change of the other, in particular, if they are connected. If they are far, we may change one trajectory of the sequence, without affecting the other. Viterbi reflects this characteristic, as it checks all combinations of neighboring (i.e. close) maneuvers to get the optimum.

We now are able to formulate our trajectory planning algorithm. We have

- the input: route points $p_1,\ldots p_n$, two orientation angles θ_1 and θ_n,
- a cost function to evaluate trajectories,
- maneuver types Π, candidates O_i and *params*(M),
- the output: a sequence of maneuvers with specified free parameters and the orientation angles $\theta_2, \ldots \theta_{n-1}$.

Fig. 2 sketches the algorithm. Starting with (p_1, θ_1) it iteratively finds optimal maneuvers to (p_i, θ_{ij}). As we have multiple intermediate angles $O_i=\{\theta_{i1}, \theta_{i2}, \ldots\}$, we keep optimal trajectory sequences to each of these angles in S. Because the number of trajectories in S only depends on the last route point (and in particular not on the route before), the runtime and memory usage is of O(n).

```
Algorithm
getOptimalTrajectorySequence
S←{(p1, θ1)};
for (i←2 to n) {
  S'←{};
  for each θij∈Oi {
    minC←∞; minS←undef;
    for each optimal sequence s∈S {
      for each maneuver type M∈Π {
        for each param∈params(M) {
          compute m that
            - is of type M
            - has param as free
                parameters
            - extends s to (pi, θij)
          if such m exists {
            s'←s extended by m;
            compute costs c of s'
                by the Evaluator
            if c<minC {
              minC←c; minS←s'; }
          } // end if exists
        }   // end for each param
      }     // end for each M
    }       // end for each s
    if minS≠undef {S'←S'∪{minS}; };
  }         // end for each θij
  S←S';
}           // end for i
if S={}  return no trajectory found
  else  return first element of S
```

Fig. 2. The trajectory planning algorithm

For a new route step p_{i+1}, we again have to check multiple orientation angles of O_{i+1}. For this, we try to extend all trajectory sequences in S by maneuvers. I.e. we compute all possible maneuver types in Π with possible *params* to get to $(p_{i+1}, \theta_{i+1,j})$. We store the optimal trajectory sequences to each of the orientation angles in S' that forms the S for the next iteration.

In the last step, $O_n=\{\theta_n\}$, i.e. we only have to check the single target angle. Thus, if there is at least one trajectory sequence found, we have $|S|=1$ and the optimal sequence is the single element of S.

3.4 Examples and Evaluation

We implemented the approach on our Carbot robot ([18], Fig. 3). It has a size of 35 cm x 40 cm x 27 cm and a weight of 4.9 kg. It is able to run with a speed of 31 cm/s. The wheel configuration allows to independently steer two wheels (Fig. 3 right). It is possible to drive curves like a car, but also to turn in place.

For curves, both the different numbers of revolutions of the powered front wheels as well as the steering angles of the steered rear wheels are adapted to follow the respective curve geometry.

To recognize the environment, Carbot's camera is able to capture images in driving direction – the respective processing is executed onboard. A 360° Lidar sensor on top is able to recognize obstacles with cm resolution. In addition, Carbot has tactile and ultrasonic sensors. The entire world modeling, including SLAM-correction of internal poses is performed on the robot. We use a Raspberry Pi (3 Model B) to execute these tasks.

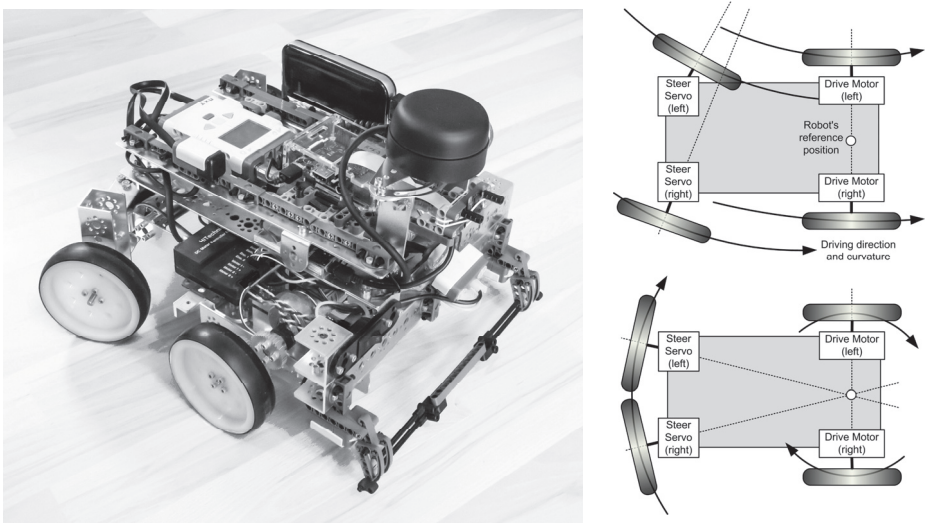

Fig. 3. The Carbot

The Carbot platform also comes with a simulation environment that is able to simulate the Carbot on sensor- and motor-level, including appropriate error models and driving effects such as slippery floors. Software binaries can be tested in the simulator without the requirement to re-compile them for the real robot. One important benefit: we can quickly construct very complex environments that would be costly to create in reality.

We created a number of environments to test our trajectory planning. One is a complex environment with walls, pillars and triangular obstacles; further environments are rectangular labyrinths. Prior to the trajectory planning, we performed a route planning that creates the route points. Currently, Carbot is able to execute route planning tasks with three approaches:

- a grid-based approach called *GAA* (*Grid-based A* Advanced*),
- based on *Extended Voronoi Diagrams*,
- based on *Visibility Graphs*.

The examples (Fig. 4) show trajectories planned by our approach whereas the route planning used GAA. Note that, even the Carbot is able to turn in place, we assigned higher costs for them, thus the planning algorithm tries to find arc maneuvers instead. The top and middle images are created by our simulation tool. The bottom images show real runs using the Lidar device – we used our debugging environment to paint the results.

To measure the performance, we randomly distributed 23 start/target positions and processed the $23 \cdot 22 = 506$ routes between these positions and plan trajectories by our approach. We used two execution environments to measure the runtime:

- a desktop PC with i7-4790 CPU, 3.6 GHz that represents a high computational power (e.g. of future robot platforms),
- a Raspberry Pi 3 Model B, 1.2 GHz 64-bit ARMv8 CPU that represents the current reduced computational power on the real Carbot.

Table 2 presents the major results. The numbers indicate the average time to compute the trajectories for a single route step.

Table 2. Execution statistics

Amount per route step	All		Reduced	
	Desktop	Carbot	Desktop	Carbot
Exec. time without cost function (ms)	0.542	8.86	0.0297	0.274
Cost function computation (ms)	0.336	5.53	0.0121	0.125
Total exec. time (ms)	0.878	14.39	0.0418	0.399
Avg. checked variations	137.7		11.56	

An amount of approx. 30-40% is spent for evaluating a trajectory by the cost function. Here, a time consuming part is to check collisions of trajectories with obstacles.

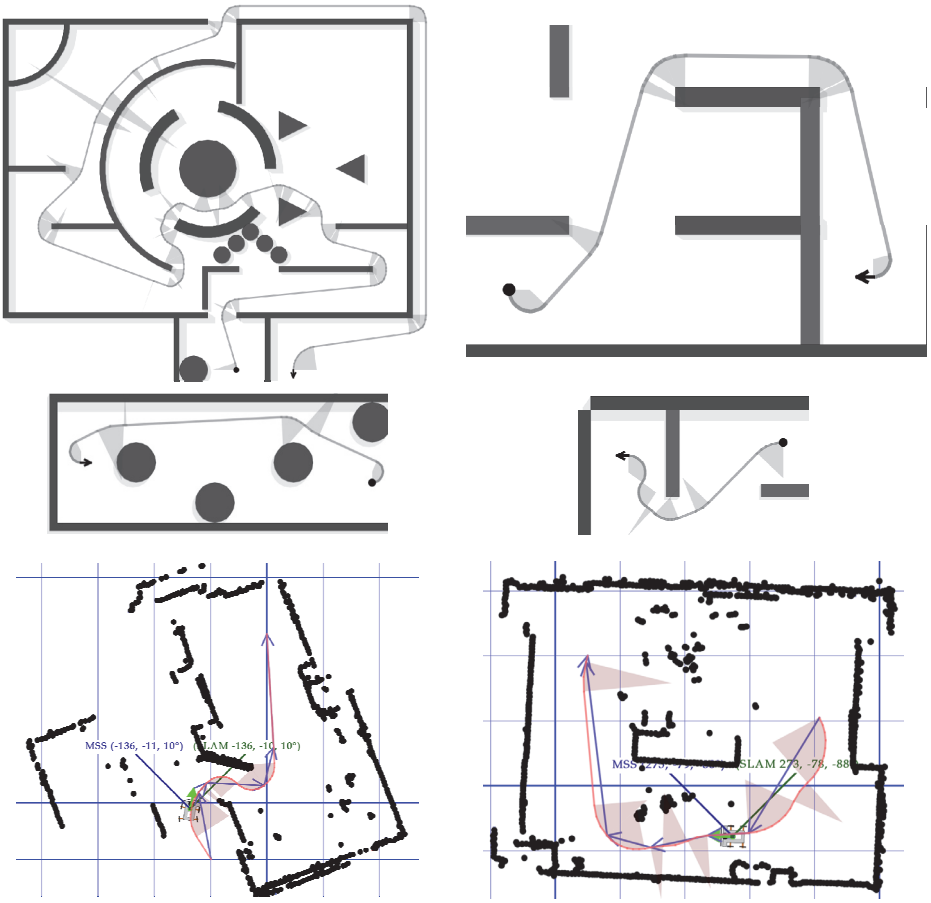

Fig. 4. Examples for planned trajectory sequences
(top, middle: simulated environments, bottom: real environments)

We divided the statistics in two packages. 'All' computed the set of candidates as introduced before, i.e. 8 maneuvers, 3 candidates for arc radii and the 5 candidates for orientations. Here, the algorithm checked 137.7 variations on average per route step.

A second package 'Reduced' works with a reduced set of candidates: only the maneuvers 2-Turns and ∫-Arcs, only a single arc radius, only three orientations and only a single arc sense of rotation were considered. As a result, the average number of variations is significantly reduced to 11.56. As only few patterns were checked, the resulting trajectories usually were much simpler. This illustrates: we may adapt the approach to reduced computations platform in a fine-grained manner, if required.

There exists another approach to deal with lower computational power. The algorithm keeps $|O_i|$ optimal variations for an iteration, but the leading maneuvers in S converge very fast to a single one. In our experiments, in 44.7% of all cases, the first maneuver for all considered sequences in S was fixed, if the algorithm checked route point 3. For route point 4, the first maneuver was fixed in 88.5% of all cases, for route

point 5, 95.7%. In our experiments, not more than 6 route points were required to fix the first maneuver. On average 3.7 route points were required.

We make use of this observation: if we had to plan long routes, we iterate through the route points until the first maneuver was fixed, usually in constant time – according to our experience this takes 3-6 routing points. We then could start driving this maneuver; meanwhile, we compute the next one.

This approach significantly reduces the computation time to find the first driving command. Usually, during driving, the robot has to react to position correction by a SLAM component or detects new obstacles that may affect further maneuvers anyway. Thus, it is not reasonable to plan all trajectories to the target in a single step.

4 Conclusions and Future Work

Our approach computes a sequence of trajectories that minimize costs. To reduce the complexity of the problem, we first compute optimal routes that only take into account the space dimensions, ignoring nonholonomic constraints. The key idea is then to produce a sufficiently large set of suitable trajectory candidates that undergo an evaluation by the cost function. The candidate set is generated by a list of maneuvers – for each we know a closed solution to map start and target configuration to a sequence of driving commands. To reduce the overall number of checked permutations, we apply a Viterbi-like mechanism. The approach is efficient. The computation time only linearly increases with the route length, whereas the first trajectory usually is available in constant time. There exist sets of parameters that allow execution even on small computing platforms.

For future work, we want to address a problem: As the Viterbi-like approach does not check all combinations, we have to accept a suboptimal trajectory sequence. This usually is tolerable. However, in rare cases a solution requires a non-optimal decision *in the middle* of the route. As Viterbi tries to optimize step by step, such paths will not be found. In the future, we want to investigate the class of scenarios that cause unacceptable solutions by our approach. A solution could artificially integrate additional route points into the optimal route or partly reverse early decisions, if such scenarios were detected.

References

1. Barraquand L., Langlois B. and Latombe J.-C. 1992. Numerical potential field techniques for robot path planning. IEEE Trans. on Syst., Man., and Cybern., 22(2), 224-241
2. Boissonnat J. D., Cerezo A. and Leblond J. 1994. A note on shortest paths in the plane subject to a constraint on the derivative of the curvature, Research Report 2160 Inst. Nat. de Recherche en Informatique et an Automatique
3. Bui, X. N., Boissonnat J. D., Soueres P., Laumond J. P. 1994. Shortest Path Synthesis for Dubins Non-Holonomic Robot, IEEE Conf. on Robotics and Autom., San Diego, CA., 2-7
4. Delingett H., Hebert M. and Ikeuchi K. 1991. Trajectory Generation with Curvature Constraint based on Energy Minimization, Proc. of the Int. Workshop on Intelligent Robots and Systems IROS91, Osaka, Japan

5. Dubins L. E. 1957. On curves of minimal length with a constraint on average curvature and with prescribed initial and terminal positions and tangents, American Journal of Mathematics, Vol. 79, No. 3, 497-516
6. Gu T. and Dolan J. M. 2012. On-Road Motion Planning for Autonomous Vehicles, ICIRA 2012, Springer, 588–597
7. Karaman S., Walter M. R, Perez A., Frazzoli E., Teller S. 2011. Anytime Motion Planning using the RRT*, IEEE Intern. Conf. on Robotics and Automation (ICRA) May 9-13, 2011, Shanghai, China, 4307-4313
8. Laumond J.-P., Jacobs P. E, Taïx M., Murray R. M. 1994. A Motion Planner for Nonholonomic Mobile Robots, IEEE Transactions on Robotics and Automation, Vol. 10, No. 5, Oct. 1994, 577-593
9. LaValle S. M.: 1998. Rapidly-exploring random trees: A new tool for path planning, TR 98-11, Computer Science Dept., Iowa State University
10. LaValle S. M. and Kuffner J. J. 2001. Randomized kinodynamic planning, Int. Journ. of Robotics Research, Vol. 20, No. 5, 378-400
11. LaValle S. M., Branicky M. S. and Lindemann S. R. 2004. Classical Grid Search and Probabilistic Roadmaps, The International Journal of Robotics Research Vol. 23, No. 7–8, July–August 2004, pp. 673-692
12. Liang T.-C., Liu J.-S., Hung G.-T., Chang Y.-Z. 2005. Practical and flexible path planning for car-like mobile robot using maximal-curvature cubic spiral, Robotics and Autonomous Systems 52, 312-335
13. Mukadam M., Yan X. and Boots B. 2016. Gaussian Process Motion Planning, IEEE International Conference on Robotics and Automation (ICRA), Stockholm, Sweden, May 16-21, 2016, 9-15
14. Muñoz V. F., Ollero A. 1995. Smooth Trajectory Planning Method for Mobile Robots, Proc. of the congress on comp. engineering in system applications, Lille, France, 700-705
15. Nelson W. L. 1989. Continuous curvature paths for autonomous vehicles, Proc. IEEE Intern. Conf. on Robotics and Automation, Scottsdale, AZ, 1260-1264
16. Pitvoraiko M. and Kelly A. 2005. Efficient constrained path planning via search in state lattices, in International Symposium on Artificial Intelligence, Robotics, and Automation in Space, 2005
17. Reeds, J.A. and Shepp, L.A. 1990. Optimal paths for a car that goes both forwards and backwards, Pacific Journ. of Math., 145, 367-393
18. Roth J. 2015. A Novel Development Paradigm for Event-based Applications, Intern, Conf. on Innovations for Community Services (I4CS), Nuremberg, Germany, July 8-10, 2015, IEEE xplore, 69-75
19. Scheuer A., Fraichard T. 1997. Continuous-Curvature Path Planning for Car-Like Vehicles, IEEE RSJ Int. Conf. on Intelligent Robots and Systems, Sept. 7-11, Grenoble, France
20. Viterbi, A. 1967. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm, IEEE Transactions on Information Theory. 13, Nr. 2, 260-269
21. Zucker M., Ratliff N., Dragan A., Pivtoraiko M., Klingensmith M., Dellin C., Bagnell J. A. and Srinivasa S. 2013. CHOMP: Covariant Hamiltonian Optimization for Motion Planning, International Journal of Robotics Research, May, 2013