Trajectory Planning for Narrow Environments That Require Changes of Driving Directions

Submitted: 19th August 2021; accepted: 8th February 2022

Jörg Roth

DOI: 10.14313/JAMRIS/4-2021/23

Abstract:
In the area of mobile robotics, trajectory planning is the task to find a sequence of primitive trajectories that connect two configurations, whereas non-holonomic constraints, obstacles and driving costs have to be considered. In this paper, we present an approach that is able to handle situations that require changes of driving directions. In such situations, optimal trajectory sequences contain costly turning maneuvers – sometimes not even on the direct path between start and target. These situations are difficult for most optimization approaches as the robot partly has to drive paths with higher cost values that seem to be disadvantageous. We discuss the problem in depth and provide a solution that is based on maneuvers, partial backdriving and free-place discovery. We applied the approach on top of our Viterbi-based trajectory planner.

Keywords: *Mobile Robots, Navigation, Trajectory Planning, Complex Turning Situations*

1. Introduction

Trajectory planning is a fundamental function of a mobile robot. When executing tasks such as transporting items, the robot has to drive trajectories that meet certain measures of optimality. Corresponding cost functions consider driving time, energy consumption, mechanical wear or buffer distance to obstacles. A planning from the current pose to a target pose takes into account an obstacle map and creates a sequence of primitive movement commands such as driving arcs or straight, whereas the resulting sequence of trajectories minimizes the given cost function. Approaches that solve this problem often have two phases:

- A *route planner* tries to find a line string in the workspace with minimal costs that does not cut obstacles, with respect to the robot's driving width,
- A *trajectory planner* in the configuration space takes the route points from the former phase, but also considers non-holonomic constraints such as minimal curve angles or driving orientations.

For route planning, there exist a variety of efficient solutions, most base on A*, where the workspace may be modelled by, e.g., grids or visibility graphs. The two-phase planning makes the problem manageable even for complex or large-scale environments as the pure workspace planning only operates on few dimensions. The second phase is much harder, because non-holonomic constraints make it difficult to use straight-forward optimization techniques. Note that other approaches, such as probabilistic path planning may compute a solution in a single phase. We discuss this in the next section.

Even though effective for many situations, sometimes the two-phase approach fails. This is because the workspace planning sometimes suggests a route that cannot adequately be mapped into the configuration space. In this paper, we consider robots that are not able to directly turn in place. If, e.g., a car-like robot has to change its driving direction (fore to backwards or vice versa), turning maneuvers are required. If starts or targets are in a narrow area, we have to plan trajectories where the locations of turns are not obvious. E.g., think of a charging station that requires the robot to drive backwards into a parking bay.

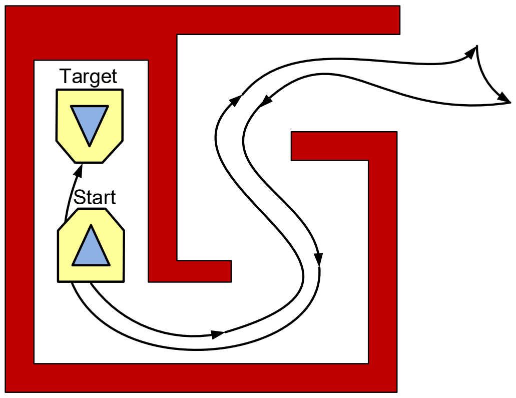

Fig. 1. A Complex Turning Situation

Fig. 1 illustrates such a situation: the robot should only slightly move, but should turn by 180°. A pure workspace planning that ignores the orientation computes a short direct route. However, if the hallway is too narrow, the robot first has to drive a costly route outside the narrow area to perform a turning maneuver.

Typical planning approaches have difficulties to find such paths. These situations can be found in everyday's life, e.g., when we want to park a car in a narrow parking bay or when we want to reverse the driving direction but have to consider constraints given by the road geometry.

This paper proposes an approach to efficiently compute paths with up two turning maneuvers. Note that for typical robots, a maximum of two changes of driving directions are sufficient. The key idea: we suggest an iterative planning that tries to optimize a sequence of *maneuvers*. As we keep multiple intermediate maneuvers per iteration, required turning maneuvers with higher local costs are also considered. To deal with paths that have to be extended to a position away from the direct connection (such as in Fig. 1), we suggest an extension of A* to find appropriate free places. We finally model all algorithmic components by a multi-strategy planning that provides a single software interface for planning tasks, meanwhile automatically applies the least complex approach for a specific scenario.

2. Related Work

Early work investigated shortest paths for vehicles that are able to drive straight forward and circular curves [3, 5, 17] or integrated additional primitive trajectories [2, 22]. Even though a combination of such primitive trajectories may appear as turning maneuvers, turning situations are not explicitly considered. Moreover, only path lengths, but not the costs for driving direction changes were examined.

Further work looked at longer paths that go through an environment of obstacles. As the variation space of possible trajectory sequences gets very large, probabilistic approaches were suitable to find at least a suboptimal solution [10, 12, 13]. They randomly connect configurations by primitive trajectories, create a connection graph, then find an actual path. Some work also used potential fields [1] or visibility graphs [15]. With the help of geometric route planners, the overall problem of trajectory planning can be reduced. In [11], the route planning step and a local trajectory planning step were recursively applied.

[16] introduced the state lattice idea, which is a discrete graph embedded into the continuous state space. Vertices represent states that reside on a hyper dimensional grid, whereas edges join states by trajectories that satisfy the robot's motion constraints. The original approach was based on equivalence classes for all trajectories that connect two states and performed inverse trajectory generation to compute the result trajectory. [7] introduced a two-step approach, with coarse planning of states based on Dynamic Programming, and a fine trajectory planning that connected the formerly generated states.

Random sampling can also be used to improve generated trajectories. E.g., CHOMP [27] used functional gradient techniques based on Hamiltonian Monte Carlo to iteratively improve the quality of an initial trajectory. The approach in [14] 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 optimized trajectories with respect to a cost function.

Special planning situations such as driving in the parking bay may explicitly be integrated into the planning component. Approaches such as [25, 26] were suitable in the area of autonomous driving, where tasks such as parking cars often occur. For this, the planning components identify the respective pattern and integrate a pre-defined movement. It is difficult to extend such an approach to arbitrary situations.

Probabilistic path planning based on RRT [12] created numerous random configurations and tried to connect them to a tree of valid trajectories. Advanced variations such as RRT* [9] introduced the property of asymptotic optimality – if we spend more runtime, the solutions get better and converge to optimal paths. This means, probabilistic approaches are in principle suitable situations that require changes of driving directions. However, they have to face two problems: first, the integration of new configurations into a tree of configurations only considers *local* optimality. The respective connections try to optimize the costs for the small part of the tree that covers the closer area of the new configuration. It is not probable that costly turning maneuvers or driving to free places are thus considered in the first place. As a result, even though an approach is asymptotically optimal, adequate routes are considered very late. The second problem: in a small scale, we still have to introduce turning maneuvers to connect new configurations. In RRT or RRT*, this problem was abstracted away by procedures called *steer* or *rewire*, however must be actually implemented to also work for complex turning situations.

3. Fundamentals

Our goal is to create a deterministic planning approach based on a two-phase planning [19].

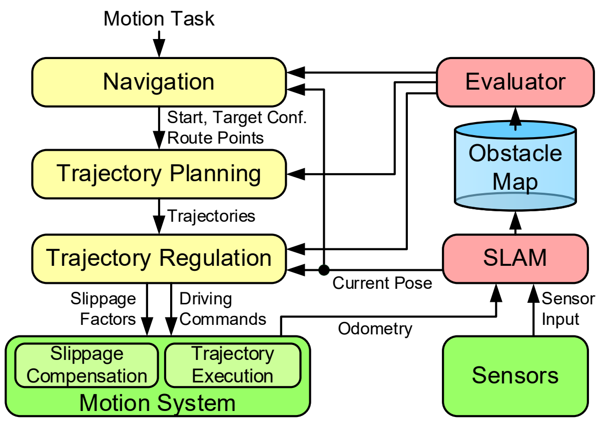

Fig. 2. Motion Planning Components

We start with the architecture of the motion planning and execution (Fig. 2). The *Navigation* component provides a point-to-point route planning in the workspace. The *Trajectory Planning* computes a drivable sequence of trajectories between configurations and considers non-holonomic constraints. The *Evaluator* computes costs of paths based on the obstacle map and the desired cost properties – a certain robot application may plug-in its own Evaluator into the system. Cost values may take into account the path length, expected energy consumption or the amount backwards driving. Also, the distance to obstacles could be considered, if, e.g., we want the robot to keep a safety distance where possible.

The lower components are not focus of this paper: The *Trajectory Regulation* permanently tries to hold the planned trajectories, even if the position drifts off due to slippage. *Simultaneous Localization and Mapping* (SLAM) constantly observes the environment and computes the most probable own position and location of obstacles by motion feedback and sensors (e.g., Lidar or camera). The current error-corrected configuration is passed to all planning components. Observed and error-corrected obstacle positions are stored in the *Obstacle Map*. The *Motion System* finally is able to execute and supervise driving commands.

3.1. Basic Considerations

We 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 the costs.

This problem has many degrees of freedom. Every two positions can be connected by an infinite number of trajectories and the problem gets worse for larger environments with obstacles. We thus introduce the following concepts:

- A route planner 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 possible set of trajectories between two route points, we only consider a small set of *maneuvers*. Maneuvers are trajectories, for which we know formulas that derive the respective geometric parameters.
- Even though these concepts reduce the problem space to a finite set of variations, this set is by far too large for complete checks. We thus apply a Viterbi-like approach that significantly reduces the number of checked variations.

We carefully separated the cost function from all planning components. We assume, we are able to assign costs to any route or trajectory sequence according to two rules: First, we must be able to define a total order on costs. This is required, as we iteratively compare routes or trajectory sequences and identify the 'best' one from a set of candidates. Second, a collision with obstacles has to result in infinite costs.

3.2. Primitive Trajectories and Maneuvers

The basic movement capabilities of a robot are defined by a set of *primitive* trajectories. The respective set can vary between different robots. E.g., the Carbot [20] is able to execute the following primitive trajectories:

- $L(\ell)$: linear (straight) driving over a distance of ℓ (that may be negative for backward);
- $A(\ell, r)$: 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, primitive trajectories specify functions that map configurations c_s to c_t. Due to non-holonomic constraints, for given c_s, $c_t \in \mathcal{C}$ there is usually 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 to 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 geometric parameters.
- For given c_s, $c_t \in C$ there exist formulas that specify the geometric 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.

Until now, we identified about thirty maneuvers of which Fig. 3 shows six. We assigned names that illustrate the maneuvers' shape, e.g., the J-Arc drives a path that looks like the letter 'J'. The Dubins-Arcs correspond to the combination of three arcs of Dubins original approach [3].

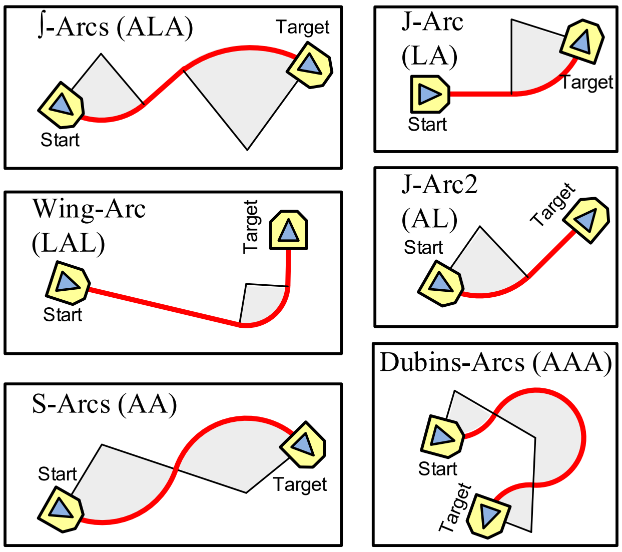

Fig. 3. Example Maneuvers

Let Π denote the set of maneuver types relevant for a certain scenario or application. Note that Π can easily be extended or reduced to reflect to robot's driving capabilities. E.g., a certain robot may not support clothoids. In this case, we could remove all maneuvers that contain a C primitive from Π.

3.3. Finding an Optimal Variation

Let $p_1=(x_1, y_1)...p_n=(x_n, y_n)$ 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 that connects start, target and all route points p_2,... p_{n-1} in-between.

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. In reality, however, it does not significantly affect the overall costs. Let O_i denote all orientation candidates for a route point $i \geq 2$. We suggest variations of angles from the previous and to the next route point (Fig. 4).

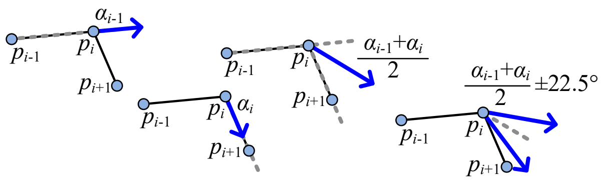
α_{i-1}
p_{i-1}
p_i
p_{i+1}
$\frac{\alpha_{i-1}+\alpha_i}{2}$
α_i
$\frac{\alpha_{i-1}+\alpha_i}{2} \pm 22.5°$

Fig. 4. Generation of Orientation Candidates

For free parameters we distinguish the small set of variations for arc senses (e.g., two 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. We suggest $\{r_{min}, 3 \cdot r_{min}, 5 \cdot r_{min}\}$, where r_{min} is the minimal curve radius. Let *params*(M) denote the set of parameters for a maneuver type $M \in \Pi$. For maneuver types with no free parameters, we define *params*(M)=$\{\varnothing\}$, whereas $\varnothing$ is an empty parameter setting.

Even though we now have a finite set of variations $\Pi \times O_i \times params(M)$ for a single route step, the total number still is 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 $2 \cdot 10^{37}$, for an average of 20 maneuvers and 5 intermediate angles. Obviously, we need an approach that computes a result without iterating through all permutations.

Our approach [19, 21] is inspired by the Viterbi algorithm [23] that tries to find the most likely path through hidden states. Our adaption looks for a sequence of maneuvers/orientations/free parameters that connects them with minimal costs.

A Viterbi-like 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. This is because a certain maneuver influences its end-orientation and thus the start-orientation of the next maneuver. 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 neighbouring maneuvers to get the optimum.

Fig. 5 illustrates the idea. Starting with (p_1, θ_1) it iteratively finds optimal maneuvers to (p_i, θ_{ij}). For the multiple intermediate angles O_i=$\{\theta_{i1}, \theta_{i2}, ...\}$, we keep optimal trajectory sequences to each of these angles in a list 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).

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 parameters to get to (p_{i+1}, $\theta_{i+1,j}$). We store the optimal trajectory sequences to each of the orientation angles in a new list S' that forms the S for the next iteration.

In the last step we have 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 get $|S|$=1 and the optimal sequence is the single element of S.

4. The New Approach

We want to extend the current two-phase approach to also support situations that require changes of driving directions. The maneuver-based trajectory planning is a solid foundation to integrate new mechanisms for required paths. Our approach can be summarized as follows:

- We integrate new turning maneuvers, required to change the driving direction.
- We extend the Viterbi-based optimization to find one or two backdriving sequences, if required.

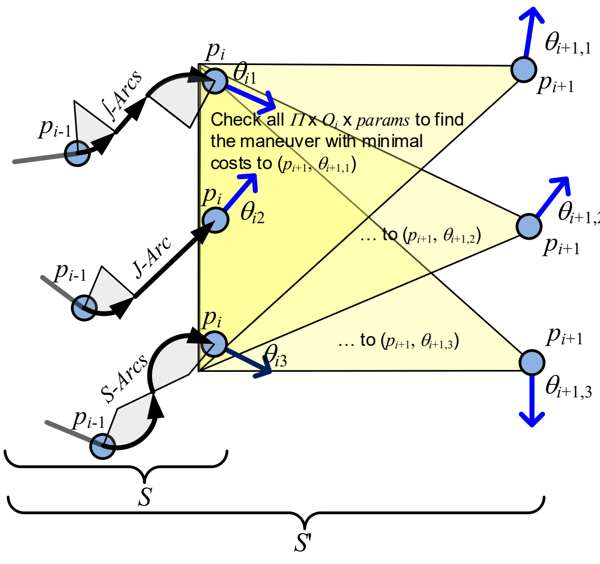

Fig. 5. Idea of Viterbi-Optimization

- We integrate a mechanism to find free places for turning maneuvers, when start and target are only connected by narrow paths.
- The new mechanisms are integrated by a software structure that supports multi-strategy planning.

Note that, whenever backdriving is required, a maximum of two backdriving sequences is sufficient for optimal paths. We come back to this point in section 4.2.

4.1. Special Turning Maneuvers

The trajectory planner prefers forward driving as long as possible, if the cost function indicates this. Sensors that map the environment or prevent collisions often point in driving direction. Thus, it is reasonable to reward forward driving. A change of driving direction requires stopping the motors that interrupts a smooth driving. A typical cost function penalizes a switch of forward to backwards driving and vice versa.

However, the change of driving directions is not always avoidable. As a first goal, we have to introduce new maneuvers that change between driving forward and backwards and vice versa. In daily life we are familiar with parking or turning maneuvers in the context of car driving that have a similar objective. For our planning approach, we identified three turning maneuvers (Fig. 6):

- *One-Bow-Turn* (*LAL*): an arc (usually with a larger turning angle) is embedded into two linear trajectories. The change of driving direction is at start or end of the arc.
- *Two-Bow-Turn* (*AAL*): two arcs with turning angle nearly 90° are connected by a change of driving direction. The linear trajectory is required to reach any target position.

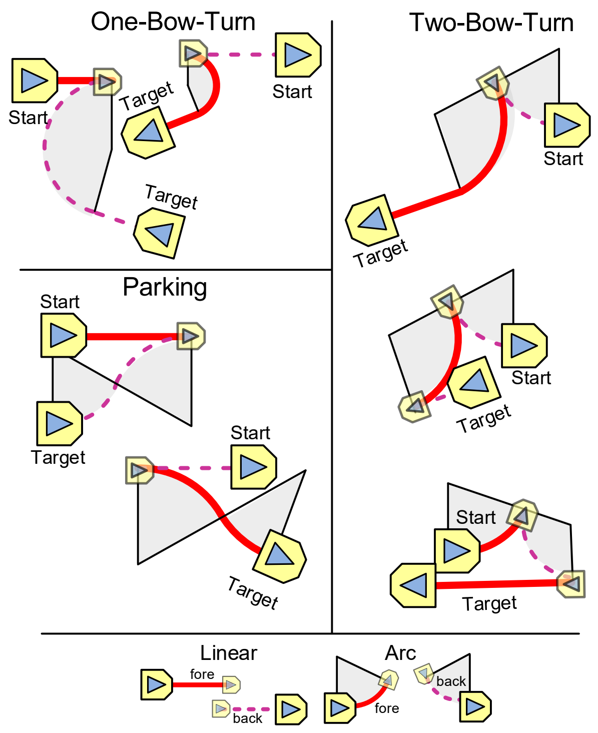

Fig. 6. New Turning Maneuvers

- *Parking (LAA)*: After a linear trajectory we have a change of drive direction. Two arcs (left/right or right/left) are then driven to reach the target position.

All these maneuvers have the single arc radius or the two arc radii as free parameters. For Two-Bow-Turn and Parking we require the same radius for both arcs to get a unique maneuver. Fig. 7 illustrates the constructions. We require the following geometric computations:

- (A) Shift the straight line through the robot's pose by *r* to left or right.
- (B) Creation of a circle (left or right) with radius *r* that touches the robot's position and has the robot's orientation as tangent.
- (C) Intersection of two straight lines.
- (D) Intersection of straight line and circle.

For the *Bow-Turn* we apply projections (A) to start and target pose. The intersection of these (C) creates the arc centre. Note that of the four variations (start or target, left or right) of the (A) projections, two do not change the driving direction. Of the remaining two, we chose the one with least costs.

For the *Two-Bow-Turn* we create the start arc using (B). We then intersect (D) a circle with radius 2*r* and the arc centre with the target straight line projection (A). The intersection is the second arc's centre. For *Parking* we apply the reverse construction: the start straight line projection (A) is intersected (D) with the target circle (B).

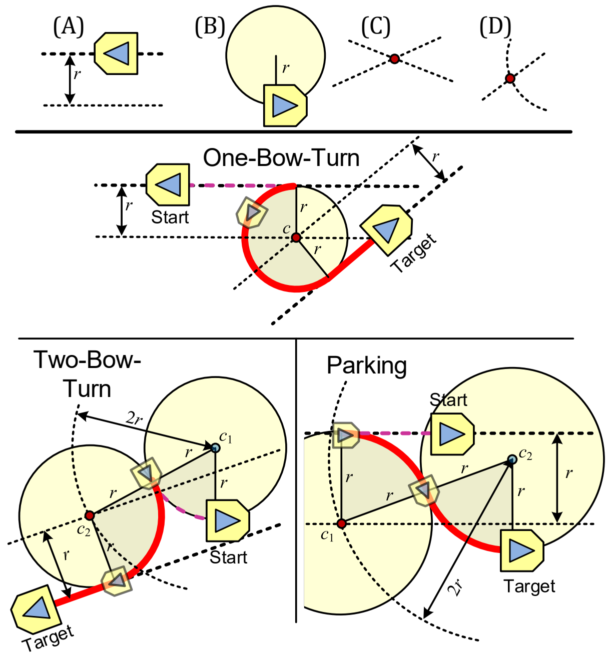

Fig. 7. Construction of Turning Maneuvers

For *Two-Bow-Turn* and *Parking* we have four variations: left/right of straight line projection and left/ right of the created start or target circle. Of these, two do not change the driving direction in the required manner. Of the remaining two, we chose the one with least costs.

Of course, all constructions (A) to (D) have to be mapped to formulas; fortunately, all have closed solutions (linear or quadratic).

4.2. Planning Backdriving

The next question is how to integrate the turning maneuvers into paths. In the following, we assume that forward driving is preferred over driving backwards. We further assume, if a robot is able to drive forward though an environment, it may also be able to drive backwards on the same trajectory. This means, the robot's geometry (e.g., concerning clearance width) is not different for the two driving directions. This is true for most robots.

Fig. 8a) shows, how turning maneuvers can be integrated into a trajectory sequence for different cases of start and target directions.

- The trajectory sequence can entirely be driven forward.
- The trajectory sequence can entirely be driven backwards – this may be a result of short route or no place for a turning maneuver.
- The trajectory sequence contains a single turning maneuver. This happens, if start and target can

only be accessed by a certain driving direction, and this is different for start and target.

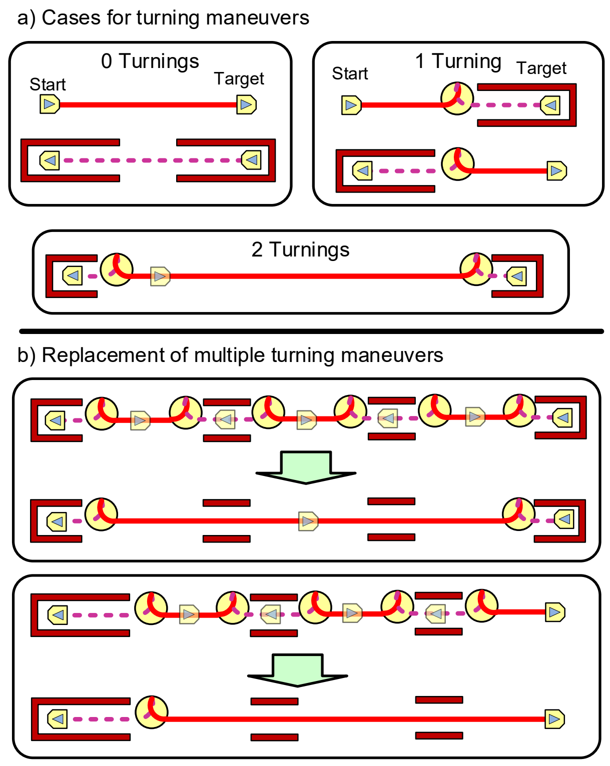
a) Cases for turning maneuvers
0 Turnings
Start
Target
1 Turning
Start
Target
2 Turnings
b) Replacement of multiple turning maneuvers

Fig. 8. Different Classes for Turning Maneuvers

- The trajectory sequence contains two turning maneuvers. This happens, if the start can only be left and target can only be reached by backwards driving. Moreover, there may be place for two turning maneuvers and the costs of forward driving pays off.

On the first glance, there may be more than two turning maneuvers. However, as a consequence of our assumptions, we get a maximum two: if we had three or more, we can always remove a sequence with an even number that start with driving backwards (Fig. 8b).

The most complex situation is presented in Fig. 9. Here, we want to switch to forward driving as soon as possible, i.e., as far as there is a place for the turning maneuver. In addition, we want to switch to backdriving as late as possible.

Our approach of Viterbi-optimization is able to create the respective trajectory sequence, if we add one modification: in addition to the orientation candidates (Fig. 4) we consider the *reverse* angles (i.e., the angles plus 180°). This modification creates sufficient candidates to integrate one or two turning maneuvers, if required.

We have to justify, why this modification in relation with the Viterbi approach creates the turning maneuvers at optimal places. Remember that the Viterbi approach iteratively goes through the workspace positions, meanwhile keeps optimal trajectory sequences to all configurations at the last considered position. I.e., at each last point, we know how to reach it in forward *and* backwards orientation (and further graded angles in-between). We have to consider two effects: 1) the switch to forward driving (if it is required in the sequence) is as early as possible, and 2) the switch to backwards driving (if it is required in the sequence) is as late as possible.

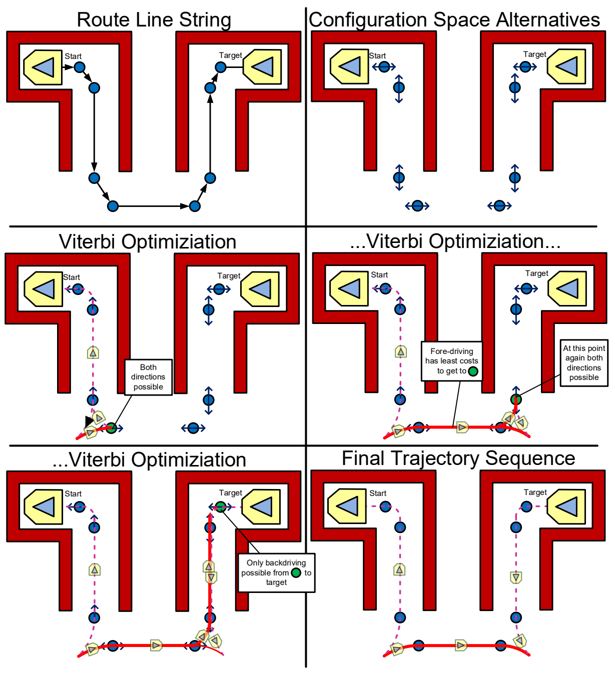
Route Line String
Configuration Space Alternatives
Viterbi Optimiziation
...Viterbi Optimiziation...
...Viterbi Optimiziation
Final Trajectory Sequence
Start
Target
Both directions possible
Fore-driving has least costs to get to ●
At this point again both directions possible
Only backdriving possible from ● to target

Fig. 9. Planning of Backdriving with Viterbi-Optimization

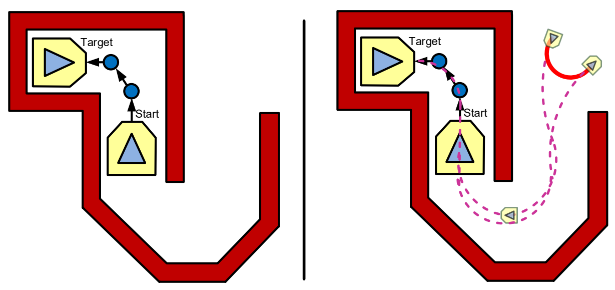

Fig. 10. Failure Situation (left) and Desired Trajectory Sequence (right)

1) This effect occurs, if the environment prohibits forward driving when leaving the starting area. This means, forward driving configurations are considered in one step, but no single maneuver is able to reach it, as long as the robot is in the narrow area. But as soon as possible, the respective turning maneuver is taken into account. In the following iterations, both driving directions are expanded simultaneously, however at a certain point, forward driving causes least costs.
2) Shortly before entering a narrow target area, multiple orientations are considered, but even backdriving orientations may be reached with least costs, if all orientations before were forward orientations. When entering a narrow area, both driving directions still are considered simultaneously. However, if the final target can only be an extension of the backdriving variation, the last possible position for changing the direction is planned.

As we can see here, we still are able to create the trajectory sequence step by step in a greedy manner without loosing optimality. As a last point, we have to discuss a more technical problem. Usually, the navigation tries to produce as few as possible route points, as the number influences the computation speed of the second planning phase. This may cause large distances between route points, if, e.g., the robot is able to drive straight in a hallway. The problem: the change of forward and backwards driving can only be planned *at* a route point computed by the navigation component. Thus, on large route sections, the planner often does not have a chance to identify an appropriate position. To solve this, we have to artificially integrate route points, if a distance exceeds a certain limit. For this, we can simply cut a large section linearly.

4.3. Free-Place Turnings

The approach until now is able to plan trajectory sequences with partial backwards driving on direct routes. However, there exist scenarios where the workspace route cannot be extended to the configuration space (Fig. 10).

This problem occurs, when a turning maneuver is required, but no route point offers enough space for its integration. This may require arbitrary long detours far away from the direct route: imagine a maze where the robot first has to drive outside to find an appropriate place for a turning maneuver.

An observation: the maximum of turning maneuvers outside the workspace route is one. This is because the situation only occurs, when both start and target are inside the same narrow area and a single turn outside this area is sufficient. Thus, we can formulate a solution as follows: find a turning position that

- offers sufficient space for a turning maneuver *and*
- generates minimal route costs for the sum of start to turning position and turning position to target.

Here, we do not want to iterate through all places, meanwhile calling the route planning two times. Our approach is based on the idea presented in [18] and extends the A* route planning approach [8].

A* Point to Point Planning. To understand our approach, we first have to briefly describe the traditional route planning based on A*. First, we have to map the environment on a graph with nodes q_i. This requires a kind of discretization. There exist several methods to do this, e.g., using a grid [4], Voronoi regions [6] or visibility graphs [24].

In a second step we have to assign cost values to edges between connected nodes, denoted $c(q_i, qj)$. We furthermore have to provide a lower cost limit estimation between two nodes, denoted $h(q_i, q_j)$. Let further $c^*(q_i, q_j)$ denote the costs of the optimal route. Our goal is now to compute $c^*(start, target)$ and the respective sequence of nodes.

The actual route computation by A* iteratively assigns three 'states' to the node: *not_visited* means, there is no knowledge how to reach it from the *start*; *open* means, we know at least one route from the start; *closed* means: we are sure how the reach a node from the start by an optimal route. Obviously, we terminate, when the target has been *closed*. To control the state changes, we need an array $g[q_i]$ that is

- ∞, if q_i is *not_visited*,
- $c^*(start, q_i)$, if q_i is *closed*,
- not less than $c^*(start, q_i)$, if q_i is *open*.

Let further denote $f[q_i]=g[q_i]+h(q_i, target)$. A* is based on a key observation: for the q_i with the lowest $f[q_i]$, we get $g[q_i]=c^*(start, q_i)$. As a result, we can assign the *closed* state to this node. In addition, we iterate through all neighbours q_j of q_i and check, if q_j can be reached with least costs going over q_i. We further assign the *open* state to all non-closed neighbours of q_i. Finally, every node gets a *backlink* entry that points to the neighbour over which a node is accessed on an optimal route from the start.

To quickly find the q_i with lowest $f[q_i]$, we need an efficient structure, e.g., the priority queue, that is based on types of ordering the *open* nodes.

Routes Through Free Places. We now want to route over a free place, sufficiently large to enable a turning maneuver. More formally, we search a free place node I that minimizes

$$c^*(start, I)+c^*(I, target) \tag{1}$$

As an additional property, the resulting costs must be below a certain cost limit, relative to the direct path. This is useful as we may consider any path that requires too much driving costs as failure and report this to the application. The developer thus defines a factor v and we only consider I with

$$c^*(start, I)+c^*(I, target) \le v\cdot opt \tag{2}$$

where $opt=c^*(start, target)$ denotes the optimal route costs. This means, the set Φ of all nodes that are candidates as free place is

$$\begin{gathered}\Phi(start, target, v) = \\ \{I \mid c^*(start, I)+c^*(I, target) \le v\cdot opt\}\end{gathered} \tag{3}$$

The remaining section describes, how we efficiently compute Φ and thus implicitly the routes from *start* to I and from I to *target*. A first observation: from

$$c^*(start, I)+c^*(I, target) \le v\cdot opt \tag{4}$$

and

$$\begin{gathered}f[q_i]=g[q_i]+h(q_i, target)=c^*(start, q_i)+h(q_i, target) \\ \le c^*(start, q_i)+c^*(q_i, target)\end{gathered} \tag{5}$$

which is true, because $h(q_i, target)\le c^*(q_i, target)$, follows

$$f[q_i]\le v\cdot opt \tag{6}$$

As the f-value of the next node in the open list cannot get smaller, we can stop, when we poll a node with an f-value larger than $v\cdot opt$. Because we do not know *opt* from the beginning, we first expand nodes (according to A* [8]) until we polled the *target* node. Then, we visit more nodes, as far as we get the first node with $f>v\cdot opt$. As a consequence, the field goes beyond the *target*. This is reasonable, as those nodes are also candidates for I. Algorithm 1 sums up these considerations and shows, how to compute the start field. To clearly distinguish the respective structures, we apply an index s to all start field arrays.

In a second round, we have to generate a target field with index t (Algorithm 2). The approach is similar to Algorithm 1, but in order to apply the appropriate ordering of route nodes, we have to incorporate these changes:

- The first *open* node is the *target*.
- Whenever we expand a node q_i, we check the distance *from* the neighbour q_j, i.e., $c(q_j, q_i)$, not *to* the neighbour.
- The estimation h is computed from *start* to the respective node.
- The sequence of *backlink* entries points to the *target* not to the *start*.

In addition, we can directly use the *opt* value of the start field. Moreover, the target field generation can benefit from a much better estimation h compared to the start field (see (*) in Algorithm 2): we set $h=g_s[q_j]$ whenever $g_s[q_j]\geq 0$. This does not change the result, but significantly reduces the number of visited nodes for the target field, thus significantly improves the runtime. Using the g_s for h does not only provide an estimation, but returns the real costs, thus is the best considerable estimation.

The start and target field generation produce $g_s[q_i] = c^*(start, q_i)$ for all q_i with $states[q_i]=closed$ and $g_t[qi]= c^*(q_i, target)$ for all q_i with $statet[q_i]=closed$. As a consequence, we are now able to provide an efficient formula for Φ:

$$\Phi(start, target, v) = \{q_i \mid states[q_i]=statet[q_i]=closed \text{ and } g_s[q_i]+g_t[q_i] \leq v\cdot opt\} \tag{7}$$

This approach is by far more efficient than an approach that iteratively computes two optimal routes for *every* node I in the graph.

In order to find an appropriate turning place, we need to evaluate the property of 'free space' for a node. As the trajectory planning is an additional step, we do not know the actual required space to drive a turning maneuver. However, we can find an appropriate simplification that can be evaluated in workspace. E.g., we can estimate the clearance space required by a turning maneuver through a node using simulations. We furthermore are able to decide, whether a node has sufficient distance to the closest obstacle. This is required anyway to create an initial graph, as a node represents a single point in workspace, but the robot has a certain geometric extent. To sum up, we are able to efficiently decide, if a certain node has sufficient distance for a turning maneuver. Let $free(q_j)$ denote this property.

We now can put all together. We compute

$$I_{opt} = \underset{I\in \Phi, free(I)}{\arg\min}\,(g_s[I]+g_t[I]) \tag{8}$$

For this, we first compute Φ with formula (7) sorted by $g_s[q_i]+g_t[q_i]$. We then iterate though Φ starting with the smallest value. For each we test $free(q_j)$ until we get a hit.

4.4. Multi-Strategy Planning

Until now, we considered different approaches to get from start to target, namely

- workspace route planning: with or without routing over free places, assigning different values for v.
- trajectory planning: with or without reverse angle candidates, different sets of maneuvers, with or without turning maneuvers.

This produces a large set of possible settings to compute a trajectory sequence. Because more complex approaches such as finding a free place request more

Algorithm 1. Start field generation

```
start_field(start, target, v)
$g_s[start]\leftarrow 0$; $f_s[start]\leftarrow 0$; $state_s[start]\leftarrow open$;
$openList.add(start)$;                  // add start to open
for all $q_i\neq start$ {               // initialize fields
    $g_s[q_i]\leftarrow -1$; $f_s[q_i]\leftarrow 0$; $state_s[q_i]\leftarrow not\_visited$;
}
$opt\leftarrow undefined$;              // costs for optimal route first unknown
do {
    $q_i\leftarrow openList.poll()$;    // get open $q_i$ with minimal $f_s[q_i]$
    if $opt$ is defined and $f_s[q_i]>v\cdot opt$  // this cannot be an $I$
        return $success$;               // going over $I$ is too costly: finish
    if $q_i=target$ $opt\leftarrow g_s[q_i]$;   // route to target found: set $opt$
    $state_s[q_i]\leftarrow closed$;
    for all neighbours $q_j$ of $q_i$ { // expand node according A*
        if $state_s[q_j]\neq closed$ {
            $g_{new}\leftarrow g_s[q_i]+c(q_i, q_j)$;
            $f_{new}\leftarrow g_{new}+h$;
            if $state_s[q_j]=not\_visited$ or $f_{new}<f_s[q_j]$ {
                $g_s[q_j]\leftarrow g_{new}$; $f_s[q_j]\leftarrow f_{new}$; // start$\rightarrow q_i\rightarrow q_j$ is less costly
                $backlink_s[q_j]\leftarrow q_i$;    // than the formerly stored route
                $state_s[q_j]\leftarrow open$;
                $openList.add(q_j)$;    // if already added, update $f$
            }
        }
    }
} while not $openList.isEmpty()$;
return $failure$;                       // no route at all from start to target
```

Algorithm 2. Target field generation

```
target_field(start, target, v)
$g_t[target]\leftarrow 0$; $f_t[target]\leftarrow 0$; $state_t[target]\leftarrow open$;
$openList.add(target)$;                 // add target to open
for all $q_i\neq target$ {              // initialize fields
    $g_t[q_i]\leftarrow -1$; $f_t[q_i]\leftarrow 0$; $state_t[q_i]\leftarrow not\_visited$;
}
do {                                    // Note: opt is known from start field
    $q_i\leftarrow openList.poll()$;    // get open $q_i$ with minimal $f_t[q_i]$
    if $f_t[q_i]> v\cdot opt$          // this cannot be an $I$
        return $success$;               // going over $I$ is too costly: finish
    $state_t[q_i]\leftarrow closed$;
    for all neighbours $q_j$ of $q_i$ { // Note: driving direction is $q_j\rightarrow q_i$ !
        if $state_t[q_j]\neq closed$ {  // expand node
            $g_{new}\leftarrow g_t[q_i]+c(q_j, q_i)$;  // Note: costs from $q_j\rightarrow q_i$ !
            if $g_s[q_j]\geq 0$         // (*) estimation start to $q_j$ available
                $h\leftarrow g_s[q_j]$; // i.e., the real costs, from start field, $g_s$
            else
                $h\leftarrow h(start, q_j)$;  // not available: compute $h$ yourself
            $f_{new}\leftarrow g_{new}+h$;    // Note: $h$ estimates start$\rightarrow q_j$ !
            if $state_t[q_j]=not\_visited$ or $f_{new}<f_t[q_j]$ {
                $g_t[q_j]\leftarrow g_{new}$; $f_t[q_j]\leftarrow f_{new}$;// $q_j\rightarrow q_i\rightarrow\ldots\rightarrow target$ is less costly
                $backlink_t[q_j]\leftarrow q_i$;    // than the formerly stored route
                $state_t[q_j]\leftarrow open$;
                $openList.add(q_j)$;    // if already added, update $f$
            }
        }
    }
} while not $openList.isEmpty()$;
return $failure$;                       // no route at all from start to target
```

runtime, we do not want to start with such an approach. In real scenarios, most of the planning tasks are still of a simple type. Thus, we should not spend too much planning time for these.

We want to support developers of corresponding applications to express planning tasks. A developer could define a list of rules, e.g.:

- If a planning without free places and without reverse angle candidates is successful, take the respective result.
- If not, consider reverse angle candidates.
- If this still is not successful, additionally integrate turning maneuvers.
- If this still is not successful, try the route planning through a free place in combination with reverse angle candidates and turning maneuvers.

We get even more variations, if we deal with different route planning approaches (e.g., grid, Voronoi regions or visibility graphs), different safety distances to obstacles, or different candidates for desired curve radii. To gain control over the multitude of possibilities, we suggest a software architecture that allows the developer to express rules to combine them. Our approach is based on two key ideas:

- First, we consider a pair of workspace route planning and trajectory planning as a single planner component. This takes start and target configurations (Fig. 11 left).
- Second, we hierarchically build new planning components that internally contain own planners, but their interface still appears as a single planner (Fig. 11 right).

Encapsulating the two planning phases by a single component allows us even to integrate alternative approaches for single-step planning such as probabilistic trajectory planning [9, 12].

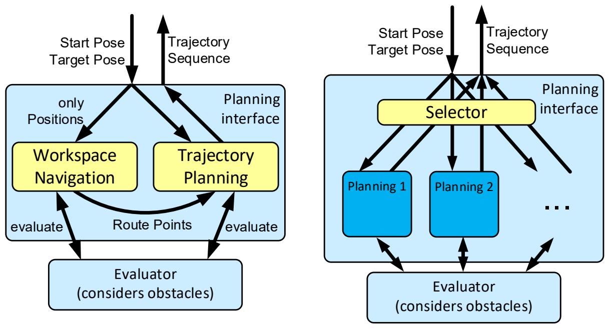

Fig. 11. Software Structure for Planning

The hierarchical components are not restricted to two levels – there can be any deep nesting. A *Selector* controls, which inner planning components may be executed and how multiple inner results are combined to a single result. Until now, we developed two hierarchical components:

- *Best-costs*: all inner components execute the planning tasks and the inner result with best costs is returned as the component's total result.
- *First-success*: the inner planning components are executed one after the other in a given order and the first successful planning is returned as the component's total result.

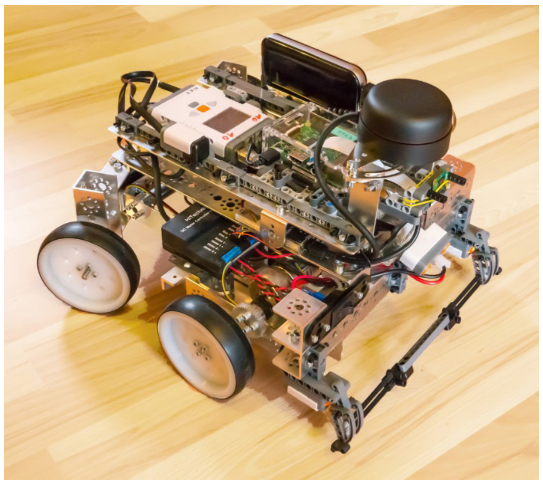

Fig. 12. The Carbot

A developer may make a choice based on available computational resources and runtime restrictions. The best-costs component requires to run all inner planning components, but this can be parallelized, if the runtime system supports it. The first-success component safes computational power, if there is a high probability for successful inner planning components that are checked first. However, the final result may be sub-optimal.

5. Experiments

We implemented the approach on our Carbot robot (Fig. 12). It has a size of 35 cm × 40 cm × 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. For arcs, 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.

A rotating Lidar device (Slamtec RPLidar A2M4) on top is used for world modelling. It scans the 360° degrees in 0.9° steps, i.e., produces 400 distance points per rotation. The sample frequency is 4000 measurements per seconds, i.e., 100 ms for every scan of 360°. The range is 0.15 to 6.00 m with a resolution of smaller than 0.5 mm.

We run the experiments in our simulation environment that simulates the robot on hardware-level [20]. It is very close to the real robot. E.g., the same binary code runs on the real robot and simulator. Motors and sensors are simulated on low levels. E.g., the simulated and real motors have the same I^{2}C command interface. Typical sensor errors and physical effects such as slippage can be applied.

We used the simulator to create situations where complex turning situations occur. Without the simulator it would be very difficult to create environments that systematically challenge the mechanisms presented above.

Fig. 13 shows characteristic environments. Situation a) is the most simple: as the required turning maneuver can be applied at the penultimate route section, the Viterbi approach checks a change of driving direction without any further mechanisms.

Situations b) and c) are more complex: a Viterbi-optimization that only considers forward driving would fail, as we need to drive backwards for a longer time to get into the parking position. Thus, we additionally require the reverse angle candidates. Note that for the entire driving in the narrow hallway, forward as well as backwards driving is considered as suitable direction. Only at the end, the forward driving candidate (and thus the corresponding trajectory sequence in the hallway) is removed and solely the presented sequences in the figure remain.

In situation d) it pays off to integrate two changes of directions. This is because, we have two places that support the turning maneuvers and their distance is large enough. Note that sequences with two turning maneuvers are only planned, if the cost function penalizes backwards driving, otherwise entirely driving backwards would be a more suitable solution.

Situations e) and f) require free-place turnings. In e), as a first try, the direct (workspace) connection is preferred over the large route around the mid square. However, any trajectory planning based on the direct connection failed, as not any turning maneuver can be integrated. Our multi-strategy planning then tries the free-place planning. The nearest free place is the area on the left. The corresponding two routes, one from start to free place and one from free place to target now can easily be planned. Situation f) is the same as situation c) apart from the start position. In f) the robot first has to fully drive outside the long hallway to integrate the turning maneuver. This situation should illustrate that in worst case, a free turning place can be far away from start and target.

In these situations, the robot already knows all involved obstacles. Either the robot has investigated the map before or an obstacle map was given. In many scenarios, however, the robot has to find a path even though the map is unknown or only partly known. In such scenarios, we have two options: either the robot first has to explore the environment or the robot applies an iterative routing approach. The next experiment illustrates the latter case. The robot should leave a maze that is unknown at the beginning. During driving, the Lidar scanner discovers new obstacles that are included into the robot's map.

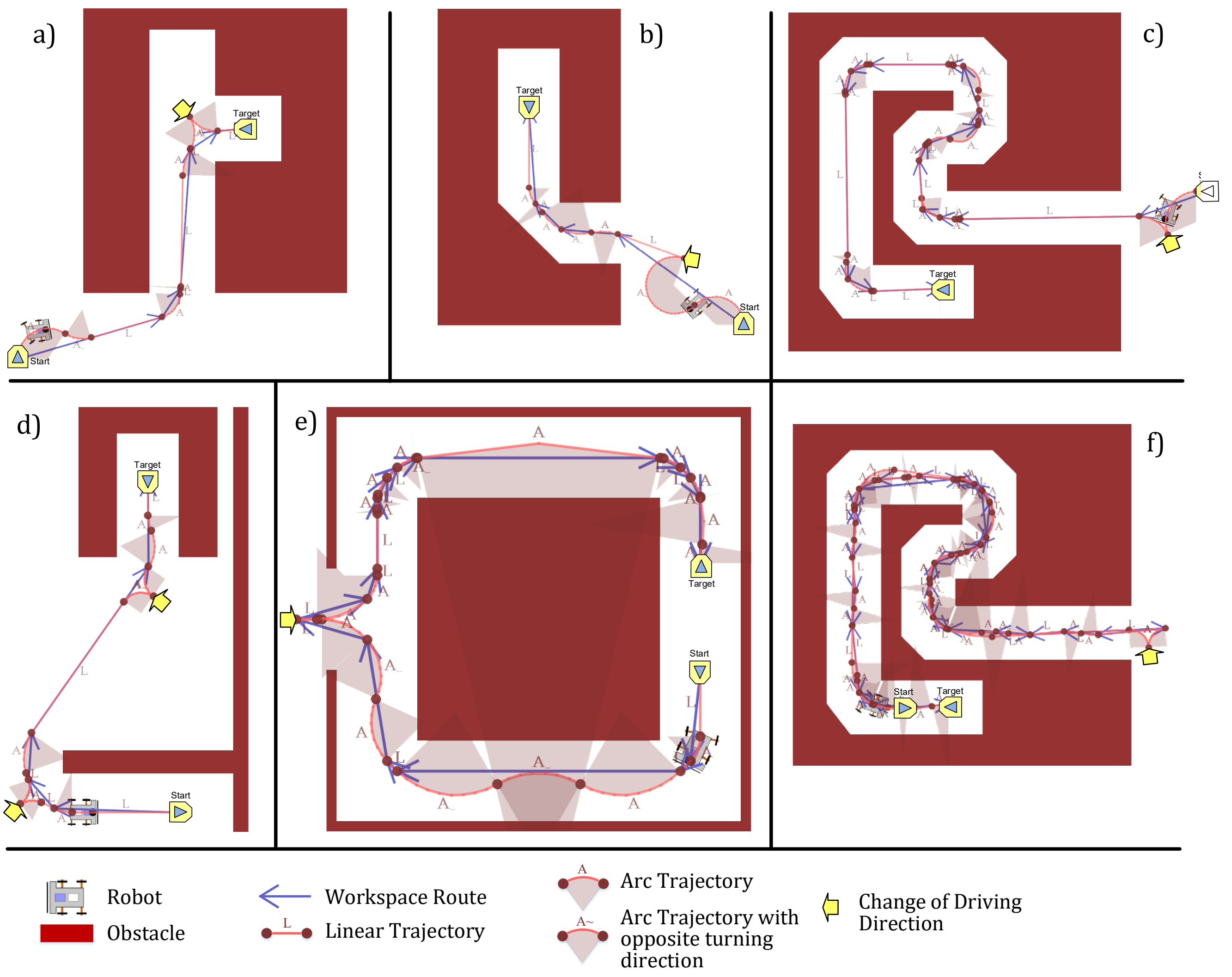

Fig. 13. Complex situations

The iterative routing approach works as follows: a trajectory sequence is planned on the partly known map, whereas undiscovered areas are for the moment considered as obstacle-free. As a result, large parts of the route go straight through currently undiscovered obstacles.

While driving, the robot permanently checks, if the formerly planned trajectories now go though newly discovered obstacles. Corresponding collisions require a new planning. To reduce the number of planning calls, we wait, until the distance between robot and collision are below a given threshold.

Fig. 14 shows some snapshots when the robot searches a way out of a narrow maze. We can easily see straight trajectories going through unknown areas. We also see the integration of turning maneuvers. This, e.g., occurs, when the robot drives into a dead end, drives backwards when leaving the dead-end and wants to continue driving forward.

Note that a route with two turning maneuvers and a route through a free place only rarely occur in the situation of iterative routing. This is because the map in the area of the target is unknown and thus all forward trajectories first can be planned without colliding with obstacles.

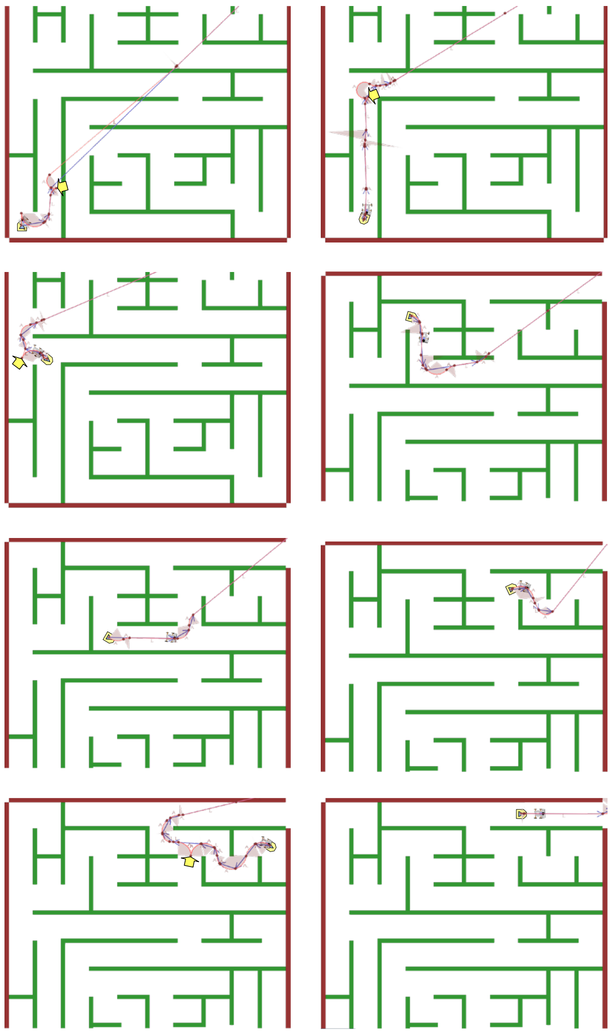

Fig. 14. Leaving a Maze

6. Conclusion

Complex turning situations that require changes of driving directions are crucial. They may lead to lengthy paths, long planning time, or even worse, a complete failure of the planning component. We suggested a solution built upon our maneuver-based planning with Viterbi-optimization.

We integrate new turning maneuvers, extend the optimization approach to create up to two backdriving sequences and are able to find free places for turning maneuvers. A software structure supports multi-strategy planning. With this, the planning component is able to react to the degree of complexity: for simple situations (still the majority), a simple and quick planning strategy still is sufficient. Only if this fails, more complex strategies are activated. The application developer can pre-define the adaptive activation of strategies in a fine-granular manner. We successfully implemented the approach on our Carbot platform.

AUTHOR

Jörg Roth – Faculty of Computer Science, Nuremberg Institute of Technology, Nuremberg, Germany, e-mail: joerg.roth@th-nuernberg.de.

REFERENCES

[1] L. Barraquand, B. Langlois and J. -C. Latombe, “Numerical potential field techniques for robot path planning”, *IEEE Trans. on Syst., Man., and Cybern.*, vol. 22, no. 2, 1992, 224–241, 10.1109/21.148426.

[2] J. D. Boissonnat, A. Cerezo and J. Leblond, “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, 1994.

[3] X. N. Bui, J. D. Boissonnat, P. Soueres and J. P. Laumond, “Shortest Path Synthesis for Dubins Non--Holonomic Robot”. In: *IEEE Conf. on Robotics and Automation*, San Diego, CA, USA, 1994, 2–7, 10.1109/ROBOT.1994.351019.

[4] M. Čikeš, M. Đakulović and I. Petrović, “The path planning algorithms for a mobile robot based on the occupancy grid map of the environment – A comparative study”. In: *XXIII Intern. Symposium on Information, Communication and Automation Technologies*, Sarajevo, 2011, 1–8, 10.1109/ICAT.2011.6102088.

[5] L. E. Dubins, “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, 1957, 497–516, 10.2307/2372560.

[6] S. Garrido and L. Moreno, "Mobile Robot Path Planning using Voronoi Diagram and Fast Marching", *Robotics, Automation, and Control in Industrial and Service*, 2015, 10.4018/978-1-4666-8693-9.

[7] T. Gu and J. M. Dolan, "On-Road Motion Planning for Autonomous Vehicles". In: Su, CY., Rakheja, S., Liu, H. (eds) Intelligent Robotics and Applications. ICIRA, 2012, 588–597, 10.1007/978-3-642-33503-7_57.

[8] P. E. Hart, N. J. Nilsson and B. Raphael, "A Formal Basis for the Heuristic Determination of Minimum Cost Paths", *IEEE Transactions on Systems Science and Cybernetics*, no. 2, 1968, 100–107, 10.1109/TSSC.1968.300136.

[9] S. Karaman and E. Frazzoli, "Incremental sampling-based algorithms for optimal motion planning", *Robotics Science and Systems*, no. VI 104.2, 2010, 10.48550/arXiv.1005.0416.

[10] S. Karaman, M. R. Walter, A. Perez, E. Frazzoli and S. Teller, "Anytime Motion Planning using the RRT*". In: *2011 IEEE International Conference on Robotics and Automation*, 2011, 4307–4313, 10.1109/ICRA.2011.5980479.

[11] J.-P. Laumond, P. E. Jacobs, M. Taïx and R. M. Murray, "A Motion Planner for Nonholonomic Mobile Robots", *IEEE Transactions on Robotics and Automation*, vol. 10, no. 5, 1994, 577–593, 10.1109/70.326564.

[12] S. M. LaValle, "Rapidly-exploring random trees: A new tool for path planning", Technical Report 98-11, Computer Science Dept., Iowa State University, 1998.

[13] S. M. LaValle and J. J. Kuffner, "Randomized kinodynamic planning", *International Journal of Robotics Research*, vol. 20, no. 5, 2001, 378–400, 10.1177/02783640122067453.

[14] M. Mukadam, X. Yan and B. Boots, "Gaussian Process Motion Planning". In: *2016 IEEE International Conference on Robotics and Automation (ICRA)*, 2016, 9–15, 10.1109/ICRA.2016.7487091.

[15] V. F. Muñoz and A. Ollero, "Smooth Trajectory Planning Method for Mobile Robots". In: *Proc. of the Congress on Comp. Engineering in System Applications,* Lille, France, 1995, 700–705.

[16] M. Pitvoraiko and A. Kelly, "Efficient constrained path planning via search in state lattices". In: *Proceedings of 8th International Symposium on Artificial Intelligence, Robotics and Automation in Space (iSAIRAS '05)*, 2005.

[17] J. A. Reeds and L. A. Shepp, "Optimal paths for a car that goes both forwards and backwards", *Pacific Journal of Mathematics*, no. 145, 1990, 367–393.

[18] J. Roth, "Efficient Computation of Bypass Areas". In: *Progress in Location-Based Services 2016, Proc. of the 13th Intern. Symposium on Location-Based Services*, Vienna, Austria, 2016, 193–210, 10.1007/978-3-319-47289-8_10.

[19] J. Roth, "A Viterbi-like Approach for Trajectory Planning with Different Maneuvers". In: *Intern. Conf. on Intelligent Autonomous Systems 15*, 2018, Baden-Baden (Germany), 2018, 10.1007/978-3-030-01370-7_1.

[20] J. Roth, "Robots in the Classroom – Mobile Robot Projects in Academic Teaching". In: *Innovations for Community Services. I4CS 2019.*, vol. CCIS 14041, 2019, 10.1007/978-3-030-22482-0_4.

[21] J. Roth, "Continuous-Curvature Trajectory Planning", *Journal of Automation, Mobile Robotics and Intelligent Systems*, vol. 15, no. 1, 2021, 9–23, 10.14313/JAMRIS/1-2021/2.

[22] A. Scheuer and T. Fraichard, "Continuous-Curvature Path Planning for Car-Like Vehicles". In: *Proceedings of the 1997 IEEE/RSJ International Conference on Intelligent Robot and Systems. Innovative Robotics for Real-World Applications. IROS '97*, 1997, 10.1109/IROS.1997.655130.

[23] A. Viterbi, "Error bounds for convolutional codes and an asymptotically optimum decoding algorithm", *IEEE Transactions on Information Theory*, vol. 13, no. 2, 1967, 260–269, 10.1109/TIT.1967.1054010.

[24] Y. You, C. Cai and Y. Wu, "3D Visibility Graph based Motion Planning and Control". In: *ICRAI '19: Proceedings of the 2019 5th International Conference on Robotics and Artificial Intelligence*, 2019, 10.1145/3373724.3373735.

[25] J. Zhang, Z. Shi, X. Yang and J. Zhao, "Trajectory planning and tracking control for autonomous parallel parking of a non-holonomic vehicle", *Measurement and Control*, vol. 53, no. 9-10, 2020, 1800–1816, 10.1177/0020294020944961.

[26] Z. Zhang, S. Lu, L. Xie, H. Su, D. Li, Q. Wang and W. Xu, "A guaranteed collision-free trajectory planning method for autonomous parking", *IET Intelligent Transport Systems*, vol. 15, no. 2, 2021, 331–343, 10.1049/itr2.12028.

[27] M. Zucker, N. Ratliff, A. Dragan, M. Pivtoraiko, M. Klingensmith, C. Dellin, J. A. Bagnell and S. Srinivasa, "CHOMP: Covariant Hamiltonian Optimization for Motion Planning", *International Journal of Robotics Research*, 2013, 10.1177/0278364913488805.