**Noname manuscript No.**
(will be inserted by the editor)

# Towards a Trajectory Planning Concept: Augmenting Path Planning Methods by Considering Speed Limit Constraints

**Lukáš Chrpa · Hugh Osborne**

Received: date / Accepted: date

**Abstract** Trajectory planning is an essential part of systems controlling autonomous entities such as vehicles or robots. It requires not only finding spatial curves but also that dynamic properties of the vehicles (such as speed limits for certain maneuvers) must be followed. In this paper, we present an approach for augmenting existing path planning methods to support basic dynamic constraints, concretely speed limit constraints. We apply this approach to the well known A* and state-of-the-art Theta* and Lazy Theta* path planning algorithms. We use a concept of trajectory planning based on a modular architecture in which spatial and dynamic parts can be easily implemented. This concept allows dynamic aspects to be processed during planning. Existing systems based on a similar concept usually add dynamics (velocity) into spatial curves in a post-processing step which might be inappropriate when the curves do not follow the dynamics. Many existing trajectory planning approaches, especially in mobile robotics, encode dynamic aspects directly in the representation (e.g. in the form of regular lattices) which requires a precise knowledge of the environmental and dynamic properties of particular autonomous entities making designing and implementing such trajectory planning approaches quite difficult. The concept of trajectory planning we implemented might not be as precise but the modular architecture makes the design and implementation easier because we can use (modified) well known path planning methods and define models of dynamics of autonomous entities separately. This seems to be appropriate for simulations used in feasibility studies for some complex autonomous systems or in computer games etc. Our basic implementation of the augmented A*, Theta* and Lazy Theta* algorithms is also experimentally evaluated. We compare i) the augmented and basic A*, Theta* and Lazy Theta* algorithms and ii) optimizing of augmented Theta* and Lazy Theta* for distance (the trajectory length) and duration (time needed to move through the trajectory).

**Keywords** Trajectory planning · Path planning · A* · Theta* · Speed limit constraints

L. Chrpa · H. Osborne
PARK Research Group
School of Computing and Engineering
University of Huddersfield, United Kingdom
E-mail: {l.chrpa, h.r.osborne}@hud.ac.uk

## 1 Introduction

Automated planning [13] belongs to one of the important research fields of Artificial Intelligence (AI). Path planning, a well known and thoroughly studied sub-field of AI planning, deals with finding paths, usually in form of polygonal chains, connecting given positions in an environment while avoiding obstacles. Trajectory planning, on the other hand, deals also with dynamic aspects such as speed since a trajectory incorporates a time dimension (i.e., position of a moving entity with respect to time). Besides spatial constraints such as avoiding obstacles a trajectory must also follow dynamic constraints (e.g. move slowly enough to pass a curve segment with high curvature) given by a particular entity. Trajectory planning is thus a crucial part of systems controlling mobile autonomous vehicles or robots. Even though trajectory planning is a part of AI planning, using general planning techniques is not appropriate in this case. It is more suitable to extend existing path planing techniques for trajectory planning because trajectory planning can be understood as a superset of path planning.

Informally, trajectory planning consists of two main areas. First, it is necessary to produce a spatial curve, a 'path', which is a result of path planning methods. Second, each point of the curve is provided with a velocity vector, which represents current speed and direction. Solutions ('paths') of the path planning problems can have many forms. The well known form is a polygonal chain, which usually results from grid based path planning [33]. Since the polygonal chain might not look realistic we might use the polygonal chain as a control polygon for computing spline curves or replace the 'faults' by arcs. On the other hand, if regular lattices are used [28], the solution is a chain of maneuvers forming the lattice. Augmenting 'paths' with velocities forms trajectories. Trajectories must follow physical and entity constraints. For instance, as a physical constraint we consider continuity of trajectory and its first derivative (velocity). As an entity constraint we may consider maximum trajectory curvature for the current speed of the entity.

Trajectory planning has many practical applications such as AgentFly [30], a system for Air Traffic Control. Another state-of-the-art system deals with trajectory planning for autonomous helicopters [32]. However, such systems do not reflect the dynamics of autonomous vehicles (such as speed limits for certain maneuvers) directly in the planning procedure because at first they produce paths (by some of the path planning methods mentioned in the following section) and then in a post-processing step they add dynamics such as velocities to produce trajectories. Such an approach, however, might not work if a path does not follow dynamics (e.g. speed of the vehicle is too high to perform a certain maneuver etc.). Moreover, such an approach might not support optimization for duration needed by a vehicle to move from starting to ending point (it may happen that longer trajectories can be passed through faster).

In this paper we present an approach which extends the current path planning methods to consider simple dynamic constraints such as speed limit constraints. We pick up on [7] (initial ideas have been introduced in [8]) where ideas of augmenting A* and Theta* algorithms to handle autonomous entities' dynamics such as speed limit constraints have been presented in an ad-hoc way. Our approach will be presented on the well-known A* [14] and state-of-the-art Theta* [23, 9] and Lazy Theta* [25] algorithms used for path planning, however, we believe that our approach can be used with other techniques (e.g RRTs) as well. Even though it seems to be sufficient to encode speed and direction as new dimensions such an approach usually causes a huge growth of the state space which must be searched. Considering intervals of speed values rather than single speed values is a promising way for reducing the search space. Two approaches, called lite and full versions, compromising

between completeness and efficiency will be discussed. The lite versions are expected to explore about the same number of nodes as basic path planning methods which is quite efficient but generally incomplete. The full versions are expected to be complete but the number of explored nodes is much higher which affects efficiency in a negative way. We use a modular architecture as a concept for implementing trajectory planning methods. It seems to be reasonable to have a four modules: representation of the environment (e.g. grids), planning method (e.g. augmented Theta*), trajectory representation (e.g. shape of the trajectory) and model of dynamics (e.g. speed limit constraints). Straightforwardly, if we plan trajectories for different entities, then only the model of dynamic should be implemented differently for each entity. For experimental purposes we implemented a very simplified model of the environment and entities' dynamics where the environment representation is based upon hexagonal or 'honeycomb'-like grids, trajectories are constructed from sequences of lines and arcs and the model of dynamics is based on basic laws of physics (e.g. uniformly accelerated motion or dependency between curvature and maximum speed).

Moreover, we extend Augmented Theta* and Lazy* Theta* (lite versions) in terms of ability to plan from/to any position, direction and interval of speed values (i.e., positions and directions are not restricted by grid). In trajectory planning optimization might be done not only for distance (i.e., the trajectory length) but also for duration (i.e., time needed for a vehicle to pass the trajectory through). Besides theoretical studies of the proposed augmented path planning methods we also provide experimental evaluation of the augmented methods, where we compare i) the augmented methods with the basic methods (velocity is added in a post-processing step) and ii) trajectory optimization for distance and duration. Furthermore, the results are thoroughly discussed showing the reasonability and justification of our approach.

The paper is organized as follows. Section 2 discusses related works in the area of path and trajectory planning. Section 3 presents existing and well known path planning algorithms. Section 4 introduces the trajectory planning terminology we use in this paper. Section 5 describes our concept of trajectory planning systems which is followed by Section 6 describing how the A*-like path planning algorithms are augmented to support the concept. Furthermore, a basic implementation of the concept is described in Section 7. Finally, we provide an experimental evaluation of our concept in Section 8. Section 9 concludes this paper and discusses some future research possibilities.

## 2 Related Works

Path and Trajectory planning is a widely studied field of AI. Many techniques and concepts have already been proposed and developed. In many cases the core consideration is how the (continuous) environment is discretized to define a graph. After that the well-known graph search algorithm A* [14] is applied.

One of the oldest techniques for path planning is based on Visibility Graphs [20]. Visibility Graphs are structures whose nodes are, besides starting and ending point, all corners of obstacles. Edges connect the nodes if and only if there is a line-of-sight between them. Paths found by this approach are optimal (shortest) in continuous 2D space. However, it has been showed that in continuous 3D space this approach does not produce optimal solutions [6].

Continuous space can be also discretized by regular geometric tessellations (usually square or hexagonal) where each cell is either blocked or free. In other words, we deal with path planning on grids [33]. The biggest advantage of this approach is its simplicity. Grids work well in environments with a smaller number of smaller obstacles. However, if

obstacles are larger, then the path planning process might become harder because often it is necessary to explore a larger area. Despite some shortcomings this technique is very popular, especially in the game industry.

Besides grids the environment can be discretized by using so called Delaunay triangulation [10]. Triangulation [26] is well-known in computer graphics. There is a parallel between the triangulation approach and Visibility Graphs. All corners of (polygonal) obstacles are also connected by lines (if there is a line-of-sight) but these lines is used for delimiting triangles. Then, the Triangle graph, in which we look for paths, is defined in such a way that nodes are triangles and edges connect nodes representing adjacent triangles. The path might connect centroids of triangles (centers of triangles' inscribed circles) which does not produce optimal paths. Note that connecting centers of triangles' circumcircles forms Voronoi diagram. The approach [10] therefore connects points on triangle edges and optimality is ensures by maximum possible underestimating cost and heuristic values during the search.

In robotics it seems to be useful to use a regular lattice discretization [28]. Lattices better correspond with maneuverability of autonomous entities (vehicles), therefore paths (trajectories) produced by these approaches can be very precise. Another approach [29] addresses path planning for car-like vehicles, where trajectories must follow a continuous curvature constraint because the vehicle cannot reorient its wheel direction instantly. Attention is also given to path (motion) planning of multi-body entities [2]. As a multi-body entity we can understand, for instance, a car with a trailer.

Motion planning had an important role in the DARPA Urban Challenge [11] by utilizing a trajectory planning approach [15] for wheeled mobile robots based on numerical optimization which considers also effects of terrain and robots' dynamics. A trajectory planning approach in multi-vehicle and dynamic environment (e.g. moving obstacles) which is also based on numerical optimization is provided in [1]. SIPP [27] is a path (trajectory) planning method for environments with moving obstacles. SIPP is based on the A* algorithm [14] which in contrary to similar approaches introduces safe intervals, time periods for configurations without any collisions. Using safe intervals can basically eliminate the necessity for considering time as an additional dimension. Anytime version of SIPP has been recently introduced in [22].

As mentioned before a basic and well known technique is an application of the A* algorithm [14] over a graph created by one of the above approaches. However, paths found by the A* usually look unrealistic, especially in grid based path planning since only centers of adjacent (or diagonally adjacent) cells in the grid can be connected. To provide more realistic paths it is necessary to connect also more distant (non-adjacent) cells. This idea has been presented in [3] where paths found by the A* are 'smoothed' (i.e., more distant cells are directly connected) in a post-processing step by iterative checking whether a node can be connected by line (i.e., the line does not intersect any obstacle) with its grand-predecessor. A state-of-the-art path planning technique Theta* [23,9] incorporates the previously mentioned idea directly in the search. I.e., it checks in the node expansion phase whether a neighbor of the expanded node can be connected by line with a predecessor of the expanded node. Lazy Theta* [25], contrary to Theta*, postpones line-of-sight checks until they are necessary. Both Theta* and Lazy Theta* will be described more thoroughly later in the text. The Field D* algorithm [12] which is also not restricted on connecting only adjacent cells but according to results published in [23] is usually slower than (Lazy) Theta*. Anytime D* [19] is an anytime and incremental search algorithm, a variant of the A* algorithm. The advantage of this approach is that the algorithm is iteratively improving the found solution as well as re-plans if necessary. Phi* [24] is an incremental variant of Theta* allowing re-planning if necessary.

It is good to mention that some of the recent works in this area use Rapidly-exploring Random Trees (RRT) [5] (most recently [31]) which are often more efficient but less optimal than the above methods. RRTs are constructed in such a way that a (random) position in space is added by connecting it to the closest position which is already in the tree. RRTs are usually used as a component that can be used for splitting a path planning task into smaller ones (then common path planning techniques are applied). The Probabilistic Roadmap planning method [17] is based on a similar idea but when a (random) position is added it is tested whether it can be connected to nearby positions (often more than one). A combination of both (RRTs and Probabilistic Roadmaps) can be found in [32]. The RRT* algorithm [16] overcomes the issue with less optimal solutions, which is the main shortcoming of RRTs. RRT* follows the asymptotic optimality property, which guarantees convergence to optimal solutions (paths). RRT* is an anytime algorithm, i.e., it can provide a feasible solution quickly and then improve it. Also it has been empirically showed that RRT* does not require significantly more computational time than RRTs. RRT* has been successfully applied in time-optimal trajectory planning for racing cars.

## 3 Path Planning

As mentioned before many concepts deal with the representation of (continuous) space as a graph-like structure and then by applying (informed) search methods (such as A*) on the structure to refine a solution (path).

### 3.1 Representation

There are many possibilities for representing the discretized environment (e.g. grids, regular lattices, triangulations etc.), though all of them can be encoded by graphs. Our terminology is therefore based on the graph theory [21].

A *directed graph* $\langle N, E\rangle$ is a structure where $N$ is a set of nodes (in literature also denoted as vertices) and $E$ is a set of edges connecting the nodes (i.e. $E \subseteq N \times N$). In path planning nodes usually stand for positions in the space and edges determine whether it is possible to directly transit between these positions. For instance, if the environment is represented by a grid, then the underlying graph is defined in such a way that nodes stand for free grid cells (obstacle-free cells) and edges are between nodes representing adjacent cells in the grid.

A *path* $p = \langle s_0, \ldots, s_k\rangle$ is a sequence of nodes ($s_0, \ldots, s_k \in N$) such that for all $i,\ 0 \leq i \leq k-1$ is the case that $(s_i, s_{i+1}) \in E$. In path planning, we look for spatial curves (hereinafter referred also as 'paths'). Finding a path from a given initial node to a given goal node in the graph representing the environment is sufficient for determining the 'path', solution of a path planning problem. For instance, if a path is found, then the 'path' is a polygonal chain connecting corresponding cells of the grid. Our following notation is derived from [23,25].
Let $next$ be a mapping $N \to 2^N$ defined in the following way:

$$next(s) = \{s' \mid (s, s') \in E\}$$

In a plain text, $next(s)$ represents a set of nodes directly accessible from a node $s$. For instance, $next(s)$ contains all nodes referring to free cells adjacent to the cell corresponding with the node $s$. Hereinafter, we denote a node from $next(s)$ as a neighbor of $s$. Let $p =$

**Algorithm 1** Skeleton of A*-like algorithms

```
1: open := close := ∅ {open is a priority queue}
2: g(s_init) := 0
3: parent(s_init) := null
4: open.Insert(s_init, g(s_init) + h(s_init))
5: while open ≠ ∅ do
6:     s := open.Pop()
7:     [SetNode(s)]
8:     if s = s_goal then
9:         return ExtractSolution(s)
10:    end if
11:    close.Insert(s)
12:    for all s' ∈ next(s) do
13:        if s' ∉ close then
14:            if s' ∉ open then
15:                g(s') := ∞
16:                parent(s') := null
17:            end if
18:            UpdateNode(s, s')
19:        end if
20:    end for
21: end while
22: return null
```

$\langle s_0, \ldots, s_k \rangle$ be a path, then $parent_p$ is a (partial) mapping $N \to N$ defined in the following way:

$$parent_p(s_i) = s_{i-1},\ 1 \leq i \leq k$$

In a plain text, $parent_p(s)$ refers to a node preceding a node $s$ on a path $p$. For example, $parent_p(s)$ refers to a node corresponding with a cell from which the entity following a path $p$ came to the cell corresponding with the node $s$. Hereinafter $parent(s)$ (without specifying a path) is used when a (partial) path is obvious for the context. For informed searches we have to define functions $g : N \to \mathbb{R}_0^+,\ h : N \to \mathbb{R}_0^+,\ c : N \times N \to \mathbb{R}_0^+$ where $g(s)$ represents an actual cost of the partial path from the initial node to $s$, $h(s)$ represents a heuristic estimation of the cost from $s$ to the goal node and $c(s, s')$ represents an actual cost of the transition from $s$ to $s'$.

### 3.2 A*

A* [14] is probably the best-known algorithm for informed search in graph-like structures. Algorithm 1 shows the skeleton of the A* and A*-like algorithms (it is same for the A*, Theta* and Lazy Theta* algorithms). Algorithm 2 refers to the A* algorithm. *open* is a priority queue containing nodes to be expanded. Nodes in *open* are ordered according to the sum of actual cost ($g$) and heuristic ($h$), i.e., a node $s'$ is placed after a node $s$ in *open* if and only if $g(s') + h(s') > g(s) + h(s)$. Hence, a node $s$ with minimum $g(s) + h(s)$ is on the top of the *open* priority queue. Note that in case that more nodes has the same $g + h$ we prioritize those added earlier into *open*. *close* is a set of already visited nodes. We begin with an initial node added into *open* (Line 4). The main loop (Lines 5-21) lasts until either a solution is found or *open* is empty (in that case there is no solution). A node

**Algorithm 2** A* algorithm

```
1: UpdateNode(s, s')
2: g_old := g(s')
3: ComputeCost(s, s')
4: if g(s') < g_old then
5:     if s' ∈ open then
6:         open.Remove(s')
7:     end if
8:     open.Insert(s', g(s') + h(s'))
9: end if
10: ComputeCost(s, s')
11: if g(s) + c(s, s') < g(s') then
12:     parent(s') := s
13:     g(s') := g(s) + c(s, s')
14: end if
15: ExtractSolution(s)
16: s' := s
17: path := ⟨⟩
18: while s' ≠ null do
19:     path.addFront(s')
20:     s' := parent(s')
21: end while
22: return path
```

**Algorithm 3** Theta* algorithm

```
1: ComputeCost(s, s')
2: if LOS(parent(s), s') then
3:     if g(parent(s)) + c(parent(s), s') < g(s') then
4:         parent(s') := parent(s)
5:         g(s') := g(parent(s)) + c(parent(s), s')
6:     end if
7: else
8:     if g(s) + c(s, s') < g(s') then
9:         parent(s') := s
10:        g(s') := g(s) + c(s, s')
11:    end if
12: end if
```

$s$ is taken from the top of the $open$ priority queue (Line 6). Procedure $SetNode$ (Line 7, in square brackets) remains empty at this stage. If $s$ is a goal node then we proceed to solution extraction (Algorithm 2), which is straightforward since we keep information about (partial) paths via $parent(s)$. Otherwise, $s$ is added to $close$ (Line 11) and its neighbors in $next(s)$ are processed (Lines 12-20). If a node $s' \in next(s)$ is in $close$, then it has already been expanded and hence we do not need to consider it once again. If $s'$ is already in $open$ but we have found a path to it with smaller cost, then we update $g(s')$ and $parent(s')$ according to this path (see ComputeCost procedure in Algorithm 2). If $s'$ is not in $open$, then $s'$ is added to $open$ (Line 8, Algorithm 2).

### 3.3 Theta*

Theta* [23,9] is a variant of A* which focuses on path planning. Theta* allows direct connection of non-adjacent nodes in the underlying graph. 'Paths' produced by Theta* are

**Algorithm 4** Lazy Theta* algorithm

```
1: SetNode(s)
2: if ¬LOS(parent(s), s) then
3:     parent(s) := arg min_{s'∈next(s)∩close}(g(s') + c(s', s))
4:     g(s) := min_{s'∈next(s)∩close}(g(s') + c(s', s))
5: end if
6: ComputeCost(s, s')
7: if g(parent(s)) + c(parent(s), s') < g(s') then
8:     parent(s') := parent(s)
9:     g(s') := g(parent(s)) + c(parent(s), s')
10: end if
```

not paths from the graph theory point of view because it allows connecting more distant nodes but 'paths' are usually shorter and more realistic than 'paths' produced by A*. In [3], a check whether a node $s$ can be directly connected by a line (a line-of-sight check) with a 'grandparent' of $s$ ($parent(parent(s))$) is iteratively performed in a post-processing step (after a path has been found by A*). In Theta* (see Algorithm 3 together with the skeleton - Algorithm 1), this idea has been applied directly in the search procedure. $UpdateNode$ and $ExtractSolution$ procedures remain the same as in Algorithm 2. If we keep only Lines 8-11 in $ComputeCost$ procedure, then we obtain the A* algorithm. A successful line-of-sight check (LOS) (Line 2), therefore, allows non-adjacent nodes to be connected. In general, LOS can be understood as a binary relation between nodes. LOS can be computed in advance and we could obtain a graph on which we could apply the A* algorithm to find a path. However, computing LOS between all the nodes might be very expensive. Theta*, therefore, works in a same way as the A* but retrospectively checks whether $s$ and $parent(parent(s))$ can be directly connected, which means bypassing $parent(s)$.

LOS can be implemented in two ways. First, a variant of the well known Bresenham's algorithm [4] can be applied (we test whether the line intersects any blocked cells). Second, if we use common geometric objects for obstacle representation, then it is possible to check whether the line intersects any of the objects. The first approach is faster in general, however, the second approach provides more realistic trajectories (the first one often avoids obstacles in rather unrealistic distances).

### 3.4 Lazy Theta*

Lazy Theta* [25] is a variant of Theta* designed for reducing unnecessary LOS checks. In Algorithm 4 it can be seen that in $ComputeCost$ procedure it is optimistically assumed that LOS is successful. After a node is selected from $open$ the $SetNode$ procedure (Line 7, Algorithm 1) verifies the assumption (the LOS check). If the assumption is incorrect then the situation must be reverted to the 'A*-like' situation, i.e., by finding an already visited node $s'$ placed next to $s$ (a possible parent of $s$) through which we reach $s$ in minimum cost (Line 3, Algorithm 4). Such a node must exist since at least one neighbor of $s$ must have been visited prior to visiting $s$. Since $s'$ has already been visited, a partial 'path' from the initial node to $s'$ is feasible. $s'$ is directly connected with $s$ in the underlying graph. Hence, the new 'path' from the initial node to $s$ is feasible as well.

### 3.5 Additional Remarks

All the algorithms presented here output sequences of nodes determining a 'path'. 'Paths', for instance, in forms of polygonal chains, must follow given constraints such as avoiding obstacles. **Soundness** of path planning methods requires that every 'path' follows the given constraints. **Completeness** of path planning methods can be determined in more ways. In general if a method is complete then the following must hold.

1. a complete method is sound (correct)
2. a method terminates (in a finite time) for every input
3. a method returns a solution if one exists or no solution if not

In path planning condition 3 can be understood in two main ways. First, we look for a 'path' in a continuous space. Second, we look for a 'path' in a discretized space. Obviously, if a space is discretized, then some 'paths' existing in a continuous space might not be found. For example, it may happen that some narrow passage between obstacles is blocked because of space discretization. Therefore, completeness of path planning methods can be determined in continuous or discretized space. For our purpose we focus on completeness in discretized space because it depends only on a method's ability to find a path in the underlying graph, which represents the environment.

It is well known that the A* algorithm is complete (if performed on a finite graph). Therefore, if the environment is represented by a graph then the path planning method using the A* to search in it is complete. The Theta* algorithm is complete as well which has been recently proven in [9]. For Lazy Theta*, soundness and completeness have not yet been formally proven. The Lazy Theta* algorithm is based on the A* and Theta* which are complete. Postponing LOS checks until it is necessary is the main difference between Lazy Theta* and Theta*. Reverting to the 'A* situation' after an unsuccessful LOS check (the $SetNode$ procedure, Algorithm 4) is the only issue which might affect completeness of Lazy Theta*.

Optimality can be understood in two ways, in the same way as completeness. First, a real optimum means the shortest 'path' in the continuous space. Second, a graph optimum means the shortest 'path' in the given graph representing the environment. For A* we are able to find the graph optimum if we use admissible heuristics. If we use Visibility Graphs to represent the environment, then we can also obtain the real optimum. In [25] it is proved that for square and cubical grids (diagonally placed cells are also considered as adjacent) the graph optima are worse than the real optima by at most $\sim 8\%$ and $\sim 13\%$ respectively. Theta* provides 'paths' that are closer to the real optima. As the authors of [23] claim Theta* does not ensure the real optimum. 'Paths' found by the Lazy Theta* can be slightly longer than the 'paths' found by the Theta*. This can be fixed by re-inserting the node (if LOS fails in the $SetNode$ procedure) to $open$ and continuing the main loop (Lines 5-21 of Algorithm 1). Such an approach is called Lazy Theta*-R and can be found in [25].

In [23, 25] empirical evaluation has shown the following. 'Paths' found by Theta* are shorter by $\sim 5\%$ in 2D and by $\sim 8\%$ in 3D that the 'paths' found by A*. 'Paths' found by the Lazy Theta are less than $0.1\%$ longer than the 'paths' found by the Theta*. A* provides solutions in about half the time of Theta*. Lazy theta* provides solutions about $30 - 50\%$ faster than Theta*. In fact, Lazy Theta*-R provides solutions at the same quality ('path' length) and almost the same running time as Theta*.

## 4 Configuration Space

Definitions and notions presented in this subsection mainly draw from [18]. We consider $A$ as an entity moving in a physical space (environment) $W$ which is represented as a subset of the Euclidean space $\mathbb{R}^D$ ($D = 2$ or 3). For simplification it is assumed in further text that an entity ($A$) is a point mass.

The following definition introduces the notions of basic and dynamic configuration and corresponding configuration spaces, which are familiar in Mechanics rather than in AI. Configuration and configuration space can be understood in a similar way as state and state space, well known terms in AI.

**Definition 1** A **basic configuration** $q_{bas}$ of an entity $A$ is a specification of the position of $A$ with respect to an environment $W$. A **basic configuration space** of $A$ $C_{bas}$ is a set of all possible basic configurations of $A$. A **dynamic configuration** $q_{dyn}$ of an entity $A$ is a specification of the position and velocity (velocity is later decomposed into direction and speed value) of $A$ with respect to an environment $W$. A **dynamic configuration space** of $A$ $C_{dyn}$ is a set of all possible dynamic configurations of $A$. ■

Note that the above definition of basic configurations differs from the definition of configurations presented in [18] by omitting entity attitude because as mentioned before entities are assumed to be point masses. On the other hand, a dynamic configuration must incorporate a direction and a speed value, forming a velocity vector describing the dynamic component.

*Remark 1* Hereinafter, a term configuration (resp. configuration space) stated without adjective (i.e., basic or dynamic) is used for an abstraction and can be replaced by terms basic or dynamic configuration (resp. basic or dynamic configuration space).

*Remark 2* Let $q$ be a configuration. Then, $pos(q)$ refers to the position of $q$, $dir(q)$ is a unit vector referring to the direction of $q$ and $spd(q)$ refers to the speed value of $q$. Note that $spd(q) \cdot dir(q)$ refers to the velocity of $q$, which takes place only if $q$ is a dynamic configuration, therefore $dir(q)$ and $spd(q)$ are undefined if $q$ is a basic configuration.

The following definition introduces a configuration transition system which is inspired by a well-known (unlabeled) state transition system.

**Definition 2** A **configuration transition system** is a 4-tuple $\mathcal{T} = \langle C, T, I, G \rangle$ where:

- $C$ is a set of **configurations** (the configuration space)
- $T \subseteq C \times C$ is a **transition relation**
- $I \subseteq C$ is a set of **initial configurations**
- $G \subseteq C$ is a set of **goal configurations**

We say that $\mathcal{T}$ **has the transition** $\langle q, q' \rangle$ if $\langle q, q' \rangle \in T$. ■

Obviously, a configuration transition system can be represented by a (directed) graph. Hence, a term path in a configuration transition system which is defined below is inspired by the Graph theory [21] rather than [18].

**Definition 3** Let $\mathcal{T} = \langle C, T, I, G \rangle$ be a configuration transition system. A **path** $p = \langle q_0, \dots q_n \rangle$ in $\mathcal{T}$ is a sequence of configurations such that $\forall i \in \{0, \dots n\} : q_i \in C$ and $\forall i \in \{0, \dots n-1\} : (q_i, q_{i+1}) \in T$. A **solution path** $p = \langle q_0, \dots q_n \rangle$ in $\mathcal{T}$ is a path such that $q_0 \in I$ and $q_n \in G$. ■

Path planning problems as known in the AI community can be directly represented by corresponding basic configuration transition systems where solution paths in the systems are in fact solutions to the corresponding problems - usually the configurations are connected by lines (see Section 3). In trajectory planning, however, solution paths in dynamic configuration transition systems provide only insights into how solutions (trajectories) might look. Trajectory and trajectory planning problem are defined as follows.

**Definition 4** A **trajectory** is a continuous function from a time interval $[t, t']$ ($t \leq t'$) to (Euclidian) space $\mathbb{R}^D$ (in physics known as a position vector), i.e.:

$$\nu : [t, t'] \rightarrow \mathbb{R}^D$$

■

**Definition 5** A **trajectory planning problem** is a tuple $\Phi = \langle I, G, Z \rangle$ where $I$ is a set of initial dynamic configurations, $G$ is a set of goal dynamic configurations and $Z$ is a set of constraints. Without loss of generality let $\nu$ be a trajectory defined on a (time) interval $[t_0, t_n]$. $\nu$ is a **solution trajectory** for $\Phi$ if and only if $\mathcal{C}$ is satisfied and there exist $q_0 \in I$ and $q_n \in G$ such that $\nu(t_0) = pos(q_0)$, $\dot{\nu}(t_0) = spd(q_0) \cdot dir(q_0)$, $\nu(t_n) = pos(q_n)$ and $\dot{\nu}(t_n) = spd(q_n) \cdot dir(q_n)$. (Note that $\dot{\nu}(t)$ stands for a first derivative of $\nu$ by $t$.) ■

In the definition above a set of constraints $Z$ is used in an abstract manner. A good example of one kind of constraint is obstacles. Obstacles stand for sets of sub-spaces (sub-environments) which no entity can pass through. Solution trajectories, therefore, must not intersect any obstacle at all. Another kind of constraint is dynamic constraints (e.g. speed limits, acceleration limits etc.) which are tailored to particular entities.

Configuration transition systems can be represented by graphs where we can apply the A* or (Lazy) Theta* algorithms on (see Section 3). In path planning we use basic configuration transition systems while in trajectory planning we can use dynamic configuration transition systems. However, the number of configurations in a dynamic configuration space is much higher than the number of configurations in a basic configuration space. Therefore, we introduce an extension of the path planning methods to support trajectory planning while trying to keep complexity on the similar level as in path planning.

## 5 Concept of a Trajectory Planning System

In this section we present the general concept of a trajectory planning system in an abstract manner as a component model. This concept can be understood as an online processing of entities' dynamics where dynamics properties are not directly encoded in the environment representation (offline processing) but are being verified during the search. A (basic) implementation of the proposed concept is discussed later in the text.

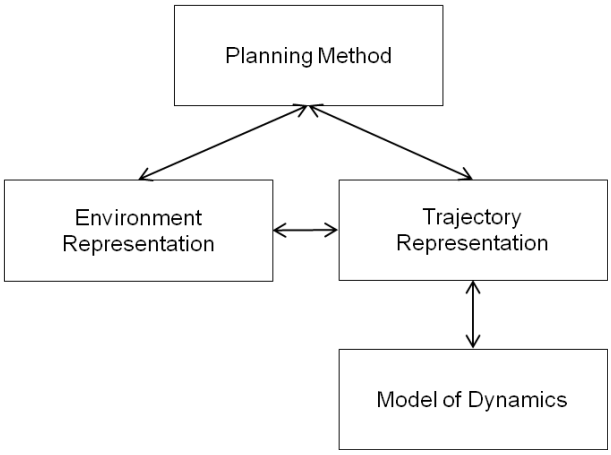


**Fig. 1** The concept of a system for trajectory planning.

The concept (see Figure 1) comes from a need for a modular architecture where each module/component has its own functionality. The *Planning Method component* incorporates an underlying planning procedure (such as Theta*) which must be augmented in terms of communicating with other components. Nodes the planning procedure deals with refer to sets of dynamic configurations having some properties in common (e.g. position and/or direction) rather than considering single ones. Determining neighbor nodes in the node expansion phase is maintained by the *Environment Representation component* representing (discretized) environments such as grids or lattices. This point is the same as in path planning, though in trajectory planning the dynamic part (e.g. speed) has yet to be determined. Speed in nodes is represented by intervals of speed values rather than single speed values (scalars). Verifying whether expanded nodes follow dynamic properties of the entity and eventually determining intervals of speed values of expanded nodes is done as follows. The *Trajectory Representation component* provides (spatial) curves ('paths') referring to a given sequence of nodes (a partial path) and the *Model of Dynamics component* determines dynamic properties of a particular entity with respect to a given curve ('path') such as a speed limit constraint (SLC), which is an interval of speed values an entity must follow prior to starting moving through the curve, and an interval of speed values the entity may have when exiting the curve. Particular components are described in detail in the following subsections.

### 5.1 Environment Representation

The purpose of the Environment Representation component is, roughly, to offer us an interface which allows the Planning Method component to navigate through the (discretized) environment and the Trajectory Representation component to create curves (spatial parts of trajectories). The component is responsible for mapping between nodes and configuration space. Bijection of the mapping (i.e., a single node represents a single configuration and vice versa) is not required which means that a single node might represent a set of (dynamic) configurations. A set of (dynamic) configurations represented by a node $s$ is denoted as $conf(s)$. This can be useful because it can make the planning process much more efficient. Therefore, we extend the definitions of the functions $pos, dir, spd$ also for nodes as

follows ($s$ is a node):

$$pos(s) = \{pos(q) \mid q \in \mathit{conf}(s)\}$$
$$dir(s) = \{dir(q) \mid q \in \mathit{conf}(s)\}$$
$$spd(s) = \{spd(q) \mid q \in \mathit{conf}(s)\}$$

It is reasonable that for every node $s$ $|pos(s)| = |dir(s)| = 1$, i.e., $s$ refers only to one position and direction, and $spd(s)$ forms an interval of values.

In our terminology the Environment Representation component provides the function $next$ (see Section 3.1) which determines neighbors of nodes. This component also provides node comparison (e.g. if a node is already presented in the $close$ list or it is a goal node) and LOS (for the (Lazy) Theta* algorithm).

## 5.2 Trajectory Representation

The Trajectory Representation component is designed to provide curves referring to particular sequences of nodes (configurations). So, the component must implement a procedure $makeCurve$ which from a given sequence of nodes generates a curve (e.g. spline) which must follow all spatial constraints such as avoiding obstacles. Generally speaking, there are many different possibilities of implementing the $makeCurve$ procedure. Generated curves are passed to the Model of Dynamics component which according to curves' properties (e.g. curvature, slope) determines ranges of possible velocities (curves accompanied with velocities form trajectories). Hence, determining of curves' properties should be computationally easy. Also, adding a new node into the sequence should not lead to generating a completely different curve. Assembling curves from segments, 'primitive' curves, referring to pairs of successive nodes in the sequence, therefore, seems to be an efficient approach (it is discussed in Section 7.2) because we need to compute only one segment after a new node is added to the sequence, which is in the node expansion phase of the planning method.

## 5.3 Model of Dynamics

The Model of Dynamics component addresses dynamic properties of entities such as autonomous vehicles, and possibly environmental properties such as terrain type. Clearly, there is a correlation between the shape of the curve and dynamic properties of particular entities.

More specifically, the Model of Dynamics serves mainly for two purposes. From a curve (or a curve segment) given by the Trajectory Representation component the component must implement a $getSLC$ procedure which is able to provide a speed limit constraint (SLC), i.e., an interval of speed values that the entity must follow prior starting moving through the curve (or curve segment). The component must also implement $reachableSpd$ procedure which given an interval of speed values of the entity starting moving through the curve provides an interval of 'reachable' speed values, i.e., the values that the entity can move at the end of the curve (i.e., when the entity leaves the curve). Basically, $reachableSpd$ considers bounds of the intervals in the sense of determining maximum possible deceleration from the lower bound of the initial interval and maximum possible acceleration from the upper bound of the initial interval. For solution trajectory extraction purposes (explained in Section 6) the component must implement $reachableSpd^{-1}$ procedure which returns the largest inverse of the $reachableSpd$ procedure, i.e.:

$$\forall \langle v, v' \rangle : \langle v, v' \rangle \subseteq reachableSpd^{-1}(curve, reachableSpd(curve, \langle v, v' \rangle))$$

**Algorithm 5** Verify procedure - verifies whether it is possible to transit from $s$ to $s'$. If so then updates speed values in $s'$

1: *verify*$(\mathbf{s}, \mathbf{s'})$
2: $curve(s, s') := trajRepr.makeCurve(s, s')$
3: $SLC(s, s') := dynModel.getSLC(curve(s, s'))$
4: $tmpspd := spd(s) \cap SLC(s, s')$
5: **if** $tmpspd = \emptyset$ **then**
6: **return** false
7: **end if**
8: $spd(s') := ReachableSpd(curve(s, s'), tmpspd)$
9: **return** true

### 5.4 Planning Method

The Planning Method component consists of an augmented version of a 'classical' path planning method (such as A* or Theta*). In this case, the basic configuration transition system does not have enough expressiveness for dynamics and using the dynamic configuration transition system might be computationally expensive. Keeping the advantage of basic configuration transition systems means that the $next$ function (provided by the Environment Representation component) deals only with the spatial part (e.g. a position and direction). The dynamic part (e.g. speed) is determined by the Model of Dynamics component. Having a (spatial) curve (provided by the Trajectory Representation component) the Model of Dynamics component can produce SLCs ($getSLC$ procedure) determining a set of speed values valid at the beginning of the curve and by using the $reachableSpd$ procedure determine a set of speed values valid at the end of the curve.

Considering the dynamic part brings additional constraints such as SLCs. In the node expansion phase ($s$ is being expanded) a new node $s'$ is selected from $next(s)$. Since we have only a spatial information about $s'$ (i.e. $pos(s')$ and $dir(s')$) we have to determine whether the dynamic part is compatible with the spatial part and, eventually, compute a dynamic information about $s'$ (i.e., $spd(s')$). It must be implemented within a *verify* procedure. Incompatibility of the spatial and dynamic part occurs when SLC is an empty interval (the curve from $s$ to $s'$ is impassable for a given entity) or disjoint with an interval of speed values in $s$ (SLC is incompatible with required speed values on the beginning of the curve). Straightforwardly, the *verify* procedure must be called after $s'$ is selected from $next(s)$. If the verify check fails $s'$ is not added to $open$ as a successor of $s$ thus we prevent to explore an inconsistent sequence of nodes.

## 6 Augmenting A*-like Algorithms

This section discusses the main contribution of this paper, that is, augmenting basic A*-like algorithms (see Section 3) to handle dynamic constraints such as speed limit constraints. This idea was introduced in [7] where augmenting A* and Theta* algorithms to support SLCs is presented in an ad-hoc way. Here, we lay the idea on the concept (see Section 5) and present the augmentation of the A*-like algorithms in a formal way including theoretical studies about soundness of augmented A*-like algorithms and completeness of the full version of augmented A*.

The idea is based on maintaining intervals of speed values which are achievable by a given entity with respect to a (partial) 'path'. The advantage of this approach is that using

intervals of speed values rather than single speed values is in reducing the necessity for handing speed as an additional dimension. An analogy can be found in SIPP [27] where safe intervals (no collision with a moving obstacle threatens in the safe interval) rather than single time-stamps have been used to reduce the search space.

Intervals of speed values are maintained as follows. For example, a vehicle goes 80km/h in the initial situation. The vehicle continues going straight ahead and after some distance its speed may range from 40km/h to 100km/h depending on whether the vehicle was decelerating or accelerating (maximum deceleration leads towards 40km/h while maximum acceleration leads towards 100km/h). Then, the vehicle might perform a turn maneuver which can be safely done if the speed of the entity is at most 50km/h, so the SLC is $\langle 0, 50\rangle$. At this point the speed of the vehicle ranges from 40km/h to 50km/h. After performing the turn maneuver the speed of the vehicle may range from 20km/h (maximum deceleration from minimum speed during the maneuver) to 60km/h (maximum acceleration from maximum speed during the maneuver). The following paragraph presents this idea in a formal way.

As indicated in the previous section a trajectory planning method such as augmented A* or (Lazy) Theta* considers also dynamic aspects (e.g. speed) during the search. The skeleton of the augmented A*-like algorithms is the same as for the basic ones (see Algorithm 1). The main difference between basic and augmented algorithms is in the node expansion phase where a node $s$, a neighbor $s'$ from $next(s)$ is selected. At this point we know the position and direction of $s'$, i.e., $pos(s')$ and $dir(s')$ (the direction of $s'$ depends on the position of both $s$ and $s'$ and thus $dir(s')$ is determined by normalizing $pos(s') - pos(s)$). $spd(s')$ is unknown at this point. Then, we have to verify whether transiting from $s$ to $s'$ follows also dynamic properties and if so, then we have to determine $spd(s')$. The implementation of the *verify* procedure is depicted in Algorithm 5. At first, $makeCurve(s, s')$ is executed to produce a curve segment $curve(s, s')$ between $s$ and $s'$ (we use an approach of assembling curves from curve segments, see Section 5.2). With $curve(s, s')$ we can determine $SLC(s, s')$, a SLC related to this curve segment. Together with $spd(s)$ which stands for a interval of speed values an entity can achieve in $s$ (if the entity is moving through the partial 'path' from an initial node to $s$) we can determine an interval of speed values (denoted as $tmpspd$) the entity can enter the curve segment $curve(s, s')$. Straightforwardly, it is an intersection of $spd(s)$ and $SLC(s, s')$ (i.e., $tmpspd = spd(s) \cap SLC(s, s')$). If $tmpspd$ is empty, then the entity cannot transit from $s$ to $s'$ at the moment and thus the verify check fails ($s'$ will not be then inserted to $open$). Otherwise $tmpspd$ is an interval of speed values the entity can have when entering the curve segment $curve(s, s')$. Then, $spd(s')$ is computed as a interval of speed values the entity can achieve when leaving the curve segment $curve(s, s')$ and thus the verify check succeeds ($s'$ will be then inserted to $open$).

Applying SLCs may cause 'gaps' in intervals of speed values throughout the sequence of nodes. For instance, if a vehicle can go between 10 and 100 km/h at some point and then it is about to perform a maneuver where the maximum speed is limited to 50 km/h, then *verify* succeeds but cuts the interval to $\langle 10, 50\rangle$ km/h before calling $ReachableSpd$. In a symbolic way, $spd(s) = \langle 10, 100\rangle$, $SLC(s, s') = \langle 0, 50\rangle$ and $spd(s') = ReachableSpd(curve(s, s'), \langle 10, 50\rangle)$. Therefore, the entity cannot go more than 50 km/h in $s$ if it is about to move to $s'$ even though the maximum reachable speed in $s$ is 100 km/h. Therefore, during the solution extraction phase we have to iteratively back-propagate the intervals of speed values as follows:

$$spd(parent(s)) = spd(parent(s)) \cap ReachableSpd^{-1}(curve(parent(s), s), spd(s))$$

**Algorithm 6** Augmented A* algorithm

```
1: UpdateNode(s, s')
2: if verify(s, s') then
3:     g_old := g(s')
4:     ComputeCost(s, s')
5:     if g(s') < g_old then
6:         if s' ∈ open then
7:             open.Remove(s')
8:         end if
9:         open.Insert(s', g(s') + h(s'))
10:    end if
11: end if
```

Knowing an interval of speed values in each node gives an insight into how speed can develop through the trajectory (see Section 7.6 for imagination) which can be used, for instance, for energy consumption optimization.

However, exact speed values (scalars rather than intervals) are needed to produce a (solution) trajectory. We can refine them from the (back-propagated) intervals, for instance, by selecting a maximum value in the each interval. In general, the exact speed values are refined as follows ($v(s)$ denotes a (single) speed value in $s$):

$$v(parent(s)) \in spd(parent(s)) \cap ReachableSpd^{-1}(curve(parent(s), s), \langle v(s), v(s)\rangle)$$

Knowing the exact speed values on the beginning and end of each curve segment is sufficient for the Model of Dynamics component which is able to interpolate the speed values throughout curve segments (e.g. by using the physical law of uniformly accelerated motion) to provide solution trajectories.

Maintaining $open$ and $close$ in the A*-like algorithms raises a question pointing to checking whether some node is already present in $open$ or $close$. In [7] two approaches were discussed as to how nodes can be distinguished, called full and lite versions. If we check whether the node is present in the $close$ or $open$ list, i.e., $s \in close$ or $s \in open$, then the check is successful if and only if there is a node $s' \in close$ or $s' \in open$ respectively such that:

- $conf(s) \subseteq conf(s')$ for the **full version**
- $pos(s) = pos(s')$ for the **lite version**

The lite and full versions compromise between completeness and complexity of the trajectory planning process. As discussed in [7] the lite version is expected to have similar complexity to basic path planning (no speed involved) at the cost of losing completeness since it may discard some alternatives leading towards solution (see Section 8.4). Hereinafter, the **basic version** denote an approach where a (non-augmented) path planning algorithm is used and dynamics (velocity) is added in a post-processing step. The theoretical aspects such as soundness and completeness of the augmented A*-like algorithms are discussed in Section 6.4.

## 6.1 Augmented A*

Algorithm 6 depicts the augmented A* algorithm. The algorithm differs from the basic (non-augmented) A* by using the *verify* procedure (Line 2), described in Algorithm 5 and discussed in the previous paragraphs. Clearly, if *verify*$(s, s')$ fails, then it is impossible to

**Algorithm 7** Augmented Theta* algorithm

```
1: ComputeCost(s, s')
2: if LOS(parent(s), s') ∧ verify(parent(s), s') then
3:     if g(parent(s)) + c(parent(s), s') < g(s') then
4:         parent(s') := parent(s)
5:         g(s') := g(parent(s)) + c(parent(s), s')
6:     end if
7: else
8:     if verify(s, s') then
9:         if g(s) + c(s, s') < g(s') then
10:            parent(s') := s
11:            g(s') := g(s) + c(s, s')
12:        end if
13:    else
14:        continue {jump to Line 12 of the skeleton algorithm (Alg. 1)}
15:    end if
16: end if
```

transit from $s$ to $s'$ (the interval of speed values is incompatible with the SLC), thus $s'$ must not be added to $open$ (as a successor of $s$). $ComputeCost$ remains the same as in the basic A* (Algorithm 2).

### 6.2 Augmented Theta*

Algorithm 7 shows the augmented Theta* (note that $UpdateNode$ procedure is the same as in (basic) A*, i.e., Algorithm 2). LOS check (Line 2) must be followed by the *verify* check because the curve segment directly going from $parent(s)$ to $s'$ is different than the curve segments going from $parent(s)$ to $s'$ via $s$. LOS in fact refers to spatial constraints (if $parent(s)$ and $s'$ can be connected by a curve segment which avoid obstacles) while *verify* check refers to dynamic constraints (an entity can move though the curve segment). If one of the tests fails, then the entity cannot directly transit from $parent(s)$ to $s'$ and thus we have to determine (the *verify* check) whether the entity can transit from $s$ to its neighbor $s'$ which is the same situation as in augmented A*.

### 6.3 Augmented Lazy Theta*

Algorithm 8 shows the augmented Lazy Theta* (note that $UpdateNode$ procedure is the same as in (basic) A*, i.e., Algorithm 2). As stated before Lazy Theta* uses a lazy evaluation paradigm to postpone LOS checks until it is necessary to perform them. Augmented Lazy Theta* uses lazy evaluation paradigm also for the *verify* check. This means that when $s'$ is a neighbor of the currently expanded node $s$ it is initially assumed that the $LOS(parent(s), s')$ and *verify*$(parent(s), s')$ checks are successful. LOS and *verify* checks are performed just after $s'$ is selected from $open$ ($SetNode$ procedure). If one of the checks fails, then the initial assumption was incorrect, thus $s$ is set as $parent(s')$ (we revert to the 'A*-like situation') in the same way as in the basic Lazy Theta* algorithm (see Algorithm 4). Then, *verify*$(s, s')$ is performed. If this also fails, then $s'$ is currently unreachable and another node must be selected from $open$ (Line 5 of the skeleton algorithm, see Algorithm 1).

**Algorithm 8** Augmented Lazy Theta* algorithm

1: $\mathbf{SetNode(s)}$
2: **if** $\neg LOS(parent(s), s) \vee \neg$*verify*$(parent(s), s)$ **then**
3: $\quad parent(s) := \arg\min_{s' \in next(s) \cap close}(g(s') + c(s', s))$
4: $\quad$ **if** $\neg$*verify*$(parent(s), s)$ **then**
5: $\quad\quad$ **continue** {jump to Line 5 of the skeleton algorithm (Alg. 1)}
6: $\quad$ **end if**
7: $\quad g(s) := \min_{s' \in next(s) \cap close}(g(s') + c(s', s))$
8: **end if**

9: $\mathbf{ComputeCost(s, s')}$
10: **if** $g(parent(s)) + c(parent(s), s') < g(s')$ **then**
11: $\quad parent(s') := parent(s)$
12: $\quad g(s') := g(parent(s)) + c(parent(s), s')$
13: **end if**

### 6.4 Theoretical Aspects

Soundness is determined by an ability of a trajectory planning method to provide solutions (trajectories) respecting all constraints in given problems (e.g. not crossing obstacles or having an appropriate speed value in every point). Completeness of trajectory planning methods can be understood in several different ways (as discussed in Section 3.5). Here completeness depends on underlying (dynamic) configuration transition systems in terms of finding solution paths between initial and goal configurations or proving their non-existence. Note that soundness is a necessary condition for completeness (as mentioned in Section 3.5).

As discussed in Section 3.5 the (basic) A* algorithm (used for path planning) is sound and complete. Therefore, if dynamic properties of entities such as speed are directly considered in the representation (i.e., we use A* to search for a solution path in a dynamic configuration transition system) then completeness (and soundness as well) remains valid. The other option, which some of the existing approaches apply, is adding dynamic properties such as speed in a post-processing step. However, if the spatial curve, the 'path', is provided by a path planning method, then it might not follow the dynamic constraints and thus the solution trajectory cannot be provided. The augmented A* algorithm, on the other hand, considers dynamic constraints (e.g. SLCs) during search and discards alternatives where dynamic constraints are violated. The formal proof of soundness of the augmented A* follows. Without loss of generality we assume correctness of implementation of the Environment Representation, Trajectory Representation and Model of Dynamics components.

**Theorem 1** *Augmented A* is sound.*

*Proof* To prove soundness of augmented A* we focus on the $UpdateNode$ procedure (Alg. 6) because the rest of the method is the same as (basic) A*. The $UpdateNode$ procedure is called for every newly opened node (the node is not present in $close$ list). The key part of the $UpdateNode$ procedure is the verify check (Line 2) which is described in Algorithm 5. Every newly opened node must pass the verify check otherwise the node is not added to $open$. *verify*$(s, s')$ firstly provides a curve (from $s$ to $s'$) and then taking into account curve properties provides $SLC(s, s')$. If the interval of speed values in $s$ ($spd(s)$) is disjoint to $SLC(s, s')$, then the entity cannot transit (directly) from $s$ to $s'$ and the verify check fails. Otherwise it is possible to transit from $s$ to $s'$. Then, $spd(s')$ must be computed which is done by applying the $ReachableSpd$ procedure provided by the Model of Dynamics component (we also assume correctness of the $ReachableSpd$ procedure). The proof of the validity of

$spd(s)$ (for every opened node) is done by induction. Firstly, $spd(s_{init})$ is valid (it is given in a problem description). In $s$ the speed of the entity can reach only values from the interval $spd(s)$. Before transiting from $s$ to $s'$ the speed of the entity must follow $SLC(s, s')$ as a necessary precondition for the transition. Therefore, if it is possible to transit from $s$ to a node $s'$ through a curve $curve(s, s')$, then at the beginning of $curve(s, s')$ the speed of the given entity is determined as an intersection of $spd(s)$ and $SLC(s, s')$. Since $spd(s)$ stands for an interval of speed values the entity have in $s$ (beginning of $curve(s, s')$) and $SLC(s, s')$ stands for an interval of speed values the entity must follow to successfully pass through $curve(s, s')$, then intersection of these intervals stands for a necessary condition for the entity to start moving through $curve(s, s')$. $spd(s')$ (i.e., the interval of speed values valid at the end of $curve(s, s')$) is then computed by the $ReachableSpd$ procedure, which is assumed to be correct.

Soundness (correctness) of back-propagation of the speed values is also proved by induction. Let $v(s)$ denote an actual speed value in $s$ (according to notation used at the beginning of this section). Selecting $v(s_{goal})$ from $spd(s_{goal}) \cap \{spd(q) \mid q \in G\}$ ($G$ is a set of goal dynamic configurations) does not cause inconsistency. Then, an actual speed in each node is computed iteratively until $s_{init}$ is reached as follows:

$$v(parent(s)) \in spd(parent(s)) \cap ReachableSpd^{-1}(curve(parent(s), s), v(s))$$

$v(s)$ is sound from assumption. Then $ReachableSpd^{-1}(curve(parent(s), s), v(s))$ provides an interval of speed values in $parent(s)$ from which $v(s)$ is reachable by moving through $curve(parent(s), s)$. $spd(parent(s))$ consists of an interval of speed values reachable from $s_{init}$ by passing all curve segments from $s_{init}$ to $s$ (see the previous paragraph). Therefore, selecting an actual speed value $v(parent(s))$ from the intersection of both the intervals is consistent. The intersection of the intervals cannot be an empty set because every speed value from $spd(s)$ has its support in $spd(parent(s))$ (i.e., $\forall v(parent(s)) \in spd(parent(s)) \exists v(s) \in spd(s) : v(s) \in ReachableSpd(curve(parent(s), s), v(parent(s)))$).

In summary, an obtained trajectory in a solution trajectory and hence the augmented A* algorithm is sound. □

This proof shows that the augmented A* algorithm expands only nodes which are consistent with dynamic constraints (besides spatial constraints), and keep updating intervals of speed values in nodes with respect to a partial 'path' leading from an initial node to them. The augmented A* algorithm therefore avoids exploring alternatives not following dynamic constraints during search in contrast of the basic A* algorithm where dynamic constraints are not considered during the search and therefore infeasible alternatives might be considered during the search. It might results in situations where solution 'paths' are not compatible with dynamic constraints. Augmented A*, on the other hand, is able to overcome this issue (fully in case of the full version of augmented A*) which is empirically showed in Section 8.

*Remark 3* Analogously to proving soundness of augmented A* we show that augmented Theta* is sound as well. We have to focus on the 'LOS part' (Lines 2-6, Algorithm 7). The rest of the algorithm is the same as augmented A*. $LOS(parent(s), s')$ ($s$ is being expanded) is followed by *verify*$(parent(s), s')$. The LOS check succeeds if a curve segment connecting $parent(s)$ and $s'$ follows spatial constraints (e.g. does not cross any obstacle). The *verify* procedure checks if the curve segment follows dynamic constraints (i.e. SLCs) and eventually update an interval of speed values $spd(s')$ (it is done analogously to the augmented A* case).

Augmented Lazy Theta* (Algorithm 8) in contrast to augmented Theta* considers that when the node is opened both $LOS$ and *verify* are assumed to be successful. This assumption is tested after the node is selected for expansion (Line 2). If either of the checks fails, then we find a node $s'$ which is a neighbor of $s$, present in $close$ and having the smallest cost (Line 3). It reverts the situation back to the 'A* case'. *verify* must be called again for $s'$ (which is now $parent(s)$) and $s$. If *verify* fails, then $s$ is unreachable in this step and cannot be expanded (i.e., another node is selected for expansion). In fact in the Lazy Theta* case we postpone the checks ($LOS$, *verify*) from the node expansion phase to the node selection phase. It is analogous to augmented Theta* since by postponing the LOS and *verify*) checks soundness is not affected.

Completeness of trajectory planning can be understood in several ways (similarly to path planning, see Section 3.5). Firstly, it refers to a possibility to find a solution trajectory in continuous space if it exists. Secondly, it refers to a possibility to find a solution path in a dynamic configuration transition system (from which a solution trajectory can be extracted) if it exists. We will consider the second way of understanding completeness of trajectory planning. Since augmented A* considers sets of configurations in a single step we have to prove that it does not miss important configurations which are part of a solution path. The idea of proving completeness of augmented A* is therefore based on the fact that if a given problem is solvable then there is always a node $s$ in $open$ consisting a configuration which is a part of some solution path. Certainly, it also depends on how nodes are distinguished (which is used for testing whether the node is already in $open$ or $close$) and we will prove in the following theorem that the full version of augmented A* is complete. Recall that full versions test if $s$ is in $open$ (or $close$) if there exists $s'$ in $open$ (or $close$) such that *conf*$(s) \subseteq$ *conf*$(s')$; lite versions, on the other hand, whether $pos(s) = pos(s')$.

**Theorem 2** *Full version of augmented A* (Algorithm 6) is complete if Lines 6-8 are omitted.*

*Proof* Augmented A* is sound (as already proven), therefore a solution found by augmented A* is valid. It is well known that (basic) A* is complete. If there exist a solution path in the underlying graph (transition system), then prior to the node expansion phase (Line 5, Algorithm 1) there is always a node in $open$ which is a part of a solution path. Augmented A* works upon a dynamic configuration transition system, however, instead of single configurations it considers sets of configurations (note that nodes in this case stand for sets of configurations). We have to prove if a given trajectory planning problem is solvable that prior to the node expansion phase there is a node $s$ in $open$ such that there is a configuration in $conf(s)$ which is a part of a solution path. We can prove this by induction. Clearly, the initial node contains a configuration with is a part of all solution paths. We therefore assume that after selecting a node $s$ for expansion consisting a configuration being a part of a solution path ($s$ is removed from $open$) at least one of its neighbor added to $open$ consists of a configuration being a part of the solution path. $next$ provides a set of all neighbor nodes of $s$ and the *verify* procedure provides them with intervals of speed values. Hence, we can deduce that these neighbor nodes cover all configurations reachable from $conf(s)$ by a single transition. Hence, there exists a newly opened node $s'$, a neighbor of $s$, being a part of the solution path. Each newly opened node is checked for the presence in the $close$ or $open$ list (Line 13 and 14). We know that the full versions are using the rule $\exists s'' \in close$ *conf*$(s') \subseteq$ *conf*$(s'')$ for checking whether $s'$ belongs to $close$. Existence of such $s'' \in close$ tells that there is no need to explore $s'$ because all the configurations *conf*$(s'')$ and hence *conf*$(s')$ have already been explored. Similarly for checking the presence of $s'$ in $open$. If

the cost of $s'$ is worse or equal than the cost of such $s'' \in open$, there is no need to add $s'$ to $open$. Otherwise, if the cost of $s'$ is better than $s''$ then $s''$ is not removed from $open$ if Lines 6-8 are omitted from Algorithm 6. If $conf(s'') \supset conf(s')$ and $s''$ is removed we might discard a node consisting a configuration that is a part of a solution path. Summarized, if a given trajectory planning problem is solvable, then prior to the node expansion (Line 5, Algorithm 1) there is a node in $open$ consisting a configuration which is a part of the solution path in the underlying dynamic configuration transition system.

If the trajectory planning problem is not solvable then similarly to basic A* the augmented A* algorithm terminates in a finite time if the underlying dynamic configuration transition system is finite as well.

Hence, the full version of augmented A* is complete. □

*Remark 4* Full versions of augmented Theta* and Lazy Theta* are not complete because we always prefer (if possible) to connect more distant cells (i.e., if LOS check is successful). Using the other way (if LOS check fails) may result into reaching a different (set of) configurations. To avoid losing completeness we have to consider both ways (i.e, add both nodes if they differ in terms of the full version to $open$).

*Remark 5* Lite versions are clearly incomplete. By a simple consideration we can see that when a node $s$ is selected, then we actually work only with $conf(s)$ but assume that we have already explored all the possible configurations which can be positioned in $pos(s)$. Clearly, we can then miss some configurations which are necessary to explore to find a solution.

For practical reasons it is necessary to discuss the running time of trajectory planners. This correlates (not absolutely) with the number of nodes which are considered during the planning process (i.e., nodes added to $open$). We raise a hypothesis, which is empirically checked (see Section 8), referring to the number of nodes considered by basic and lite versions.

**Hypothesis** Without loss of generality we assume that the cost and heuristic functions are the same for both basic and lite versions. The number of nodes considered by basic and lite versions is roughly the same ($\pm 10\%$) for an arbitrary trajectory planning problem.

Even though the hypothesis is presented in an informal manner it says that lite versions should be able to provide solutions in a similar running time to basic versions (no matter which planning method is used). This hypothesis reflects the fact that nodes are distinguished in the same way for both versions. Although there are some differences between the versions the expected solutions (trajectories or 'paths') are similar, therefore explored space should be similar as well.

However, considering full versions we have to take into account also direction and interval of speed values. For instance, if we enter a cell in a different direction than before, then it is considered as a different node thus it must be counted. Therefore, we expect that full versions consider $c|dir|$ times more nodes than basic (or lite) versions ($c$ is a factor representing an average number of considered intervals of speed values). However, full versions of augmented Theta* and Lazy Theta* seem to be unusable because $|dir|$ is very large (some experimental results have been already presented in [7]).

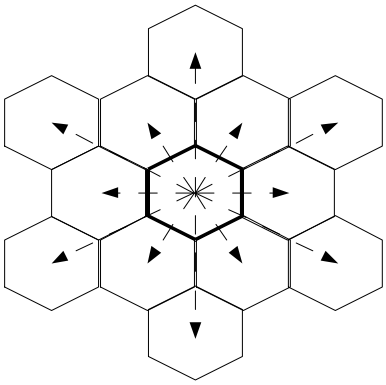

**Fig. 2** Cells considered as adjacent (also 'diagonally' placed ones).

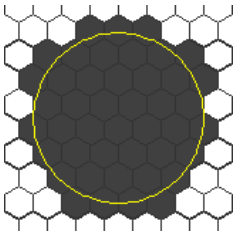

**Fig. 3** A sample obstacle.

## 7 Implementation

In the previous section we introduced augmented A*, Theta* and Lazy Theta* algorithms which implement the Planner component. This section addresses implementation details of the other components. Moreover, we discuss implementing the heuristic function ($h$) which is used to optimize either trajectory length, or duration needed by an entity to pass through the trajectory.

### 7.1 Environment Representation Component

Path (trajectory) planning algorithms usually work upon grids which somehow discretize continuous space. Commonly, grids are made from regular geometric tessellations. We use hexagonal grids where we also consider 'diagonally' placed cells as adjacent (see Figure 2). To add a 3rd dimension we have to form a honeycomb-like structure where an entity can navigate through 'flight levels'. The distribution of 'flight levels' is regular and correlated with the density of the grid (in our case distance between adjacent 'flight levels' is a half of distance between adjacent hexagonal cells).

Obstacles stand for objects or zones which must not be crossed by any trajectory. An obstacle can be defined by a set of cells which are set as blocked (we cannot open a node referring to a blocked cell). A more suitable method for representing obstacles is to use geometric objects such as circles, polygons (2D) or cylinders, cuboids, spheres (3D) etc. With geometric representations of obstacles we simply have that the cell must be blocked if it intersects any obstacle (see Figure 3).

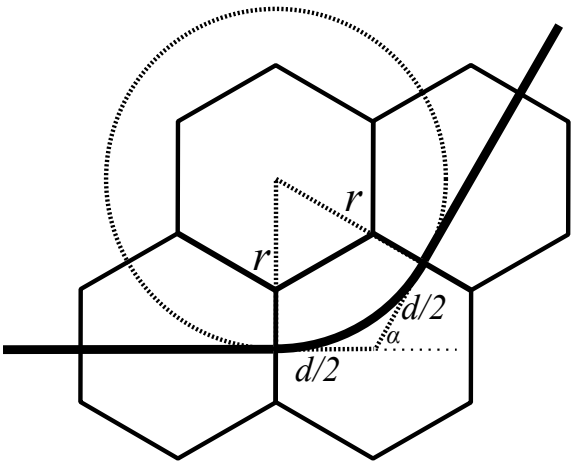


**Fig. 4** A sample turn maneuver. The thick line emphasize a part of a trajectory. $\alpha$ stands for a turn angle, $d$ stands for a distance between adjacent cells and $r$ stands for a turn radius.

The underlying (dynamic) configuration space can be defined in the following way. Configurations are positioned only in centers of free cells. For A* the configurations consider directions pointing to adjacent cells only (i.e., 12 possible directions) while for (Lazy) Theta* directions can point to any other 'visible' cell (the number of possible directions is finite if the number of cells in the grid is finite). In our implementation, speed values are discrete and range from given minimum to maximum speed values (given by the specific Model of Dynamics component).

### 7.2 Trajectory Representation Component

In our implementation we focus on the simplest possible trajectory representation. Polygonal chains are simple enough but it must be possible to change direction in a smooth way. Therefore, we use two types of atomic curve segments which are used for constructing partial curves connecting configurations. The first type is represented by (straight) lines which are used for moving between the grid cells (Straight Maneuvers). The second type is represented by arcs which are used for smooth changes of directions (Turn Maneuvers).

In 2D, the situation is easy to imagine. A sample Turn Maneuver is depicted in Figure 4. Turn angle $\alpha$ can be easily determined because the change in direction is known. Even though $\alpha$ is known there is still an infinite number of possible Turn Maneuvers. It is reasonable to keep the whole Turn Maneuver within the cell because we can be sure that the curve representing the Turn Maneuver does not cross any obstacle (the cell is not blocked). On the other hand, it is reasonable to have a turn radius as large as possible to keep the curvature as small as possible. If $d$ stands for a distance between (non-diagonal) adjacent cells, then the Turn Maneuver must start (and end) a distance of $d/2$ from the cell center. $d/2$ is a radius of hexagon's inscribed circle and therefore the arc representing the Turn Maneuver stays within the hexagon cell. Center of Turn Maneuver's circle is in the point of intersection of perpendiculars to Straight Maneuvers' lines led in $d/2$ distance from the hexagon center.

A turn radius $r$ can be determined analytically if we know $\alpha$ and $d$ in the following way:

$$r = \frac{d}{2} \cot \frac{\alpha}{2}$$

Determining the turn radius, which is a reciprocal of Turn Maneuver's curvature, is sufficient for the Model of Dynamics which determines SLCs (see Section 7.3). Note that trajectories often do not cross centers of cells, i.e., the corresponding configurations (e.g. Figure 4), but this does not violate the conditions for trajectories to be solution trajectories.

In 3D, the situation can be handled similarly (in fact we generalize the 2D case). The arc lies in the plane defined by two lines referring to preceding and succeeding Straight Maneuvers. Besides the curvature of Turn Maneuvers a slope change can be taken into account as well.

### 7.3 Model of Dynamics Component

We consider only two kinds of atomic curve segments - Straight and Turn Maneuvers (as stated in the previous subsection). It is only possible to adjust speed (accelerate or decelerate) during Straight Maneuvers. The rate of acceleration or deceleration must also be constant in any one Straight Maneuver. Practically, during a trajectory extraction phase a Straight Maneuver which connects more distant cells (Theta* or Lazy Theta* is used) is split into (at most) 3 parts (Straight Maneuvers). The first part ensures accelerating to maximum speed, the second part serves for moving at maximum speed and the third part ensures the deceleration required to fulfill, for instance, a SLC for the following Turn Maneuver. Depending on the actual situation some of the parts might be omitted (e.g. we are already moving at maximum speed at the beginning or we must begin to decelerate before reaching maximum speed etc.).

For Turn Maneuvers we model the physical law which requires the minimum turn radius to vary as the square of the speed (i.e., $r \propto v^2$). If $v_{tunit}$ stands for the maximum speed for a Turn Maneuver whose turn radius is 1 (unit), then the maximum speed for a Turn Maneuver $v_{tmax}$ whose turn radius is $r$ is:

$$v_{tmax} = \sqrt{r} \cdot v_{tunit}$$

In 3D we also need to model slopes and slope changes. Constant slopes are Straight Maneuvers and slope changes are Turn Maneuvers. A constant positive (ascending) slope causes an entity's rate of acceleration to decrease (the second derivative of its speed is negative) while a negative slope causes the entity's rate od acceleration to increase.

$$\begin{aligned} slope > 0 &\Rightarrow v'' < 0 \\ slope = 0 &\Rightarrow v'' = 0 \\ slope < 0 &\Rightarrow v'' > 0 \end{aligned}$$

We do not model acceleration changes on changing slopes ($v'' = 0$) but we prohibit Turn Maneuvers with a slope change above a given threshold.

### 7.4 Heuristics

There are two main options how trajectories can be optimized - on distance or duration. Obviously, in basic path planning the shortest 'path' means the fastest 'path' (we are moving in constant speed). However, if dynamic properties are incorporated, then the shortest trajectory may differ from the fastest one because some maneuvers on the shortest trajectory may require to be performed in significantly lower speed.

Optimization on distance can be done in the same way for both the basic and augmented (A*-like) algorithms. An actual cost of a partial trajectory from the initial configuration to a node $s$ $g(s)$ is computed as a real length of the partial trajectory, i.e., an ordered sequence of atomic curve segments. A cost between nodes $s$ and $s'$ $c(s, s')$ is computed analogously, i.e., the length of atomic curve segments connecting $s$ and $s'$. A heuristic estimation of the distance from a node $s$ to the goal configuration $h(s)$ is computed as the Cartesian distance between position of $s$ ($pos(s)$) and the position of the goal configuration. Obviously, such a heuristics is admissible because the real distance cannot be shorter than the Cartesian distance.

Optimization on duration is applicable only for augmented algorithms because the basic ones cannot deal with time. An actual cost of a partial trajectory from the initial node to $s$ $g(s)$ is computed as a time in which the partial trajectory can be passed. However, during the search we only have sequences of atomic curve segments, so it is necessary to extract (partial) trajectories from them (by calling $ExtractSolution(s)$). $c(s, s')$ is computed similarly for Theta*. In Lazy Theta* we do not know the exact interval of speed values in $s'$ ($spd(s')$) because *verify* has not yet been called and we only assume (optimistically) that transiting from $parent(s)$ to $s'$ is possible. Thus $spd(s')$ can be estimated by calling the $MakeCurve$ and $ReachableSpd$ procedures. Therefore a cost function $c(parent(s), s')$ is computed as a minimum time needed for transiting from $parent(s)$ to $s'$ but SLC is not taken into account which may underestimate the actual cost.

A heuristic estimation of the distance from $s$ to the goal node $h(s)$ can be computed as a time necessary to move directly from position in $s$ directly towards the goal configuration at maximum possible speed (i.e., the Cartesian distance from $s$ to the goal is divided by maximum entity speed). Obviously, the heuristics is also admissible.

Of course, we can optimize for a different metrics than distance and duration. A useful metric can be, for instance, energy (fuel) consumption. Optimizing energy consumption is very useful in reducing costs and also reducing pollution. To preserve an optimal (or nearly optimal) energy consumption the entity should not change its speed often but also should not move at maximum speed. Optimizing for energy consumption, we believe, might have very similar aspects as optimizing for duration. Another example of a metric can be minimizing risks which an entity undergoes during moving from initial to goal configuration. For instance, the entity could get stuck in mud, fall from the cliff edge etc. An optimal trajectory in this case should avoid such places even though its length or duration needed to pass through it might be much higher. The cost may depend on the distance the entity passes through in the 'risky' environment (e.g. mud) or how many 'risky' maneuvers the entity does. From this perspective providing an informative (and admissible) heuristic is needed.

### 7.5 Planning from/to arbitrary position, direction and speed

Discretization of continuous space provided by grids improves efficiency of the trajectory (path) planning process. However, it certainly limits starting and ending conditions such as

positions which must be related to centers of grid cells. Ability to plan a trajectory from or to arbitrary position (in continuous space), direction and speed value (or the interval) is desirable, especially for real-world applications. In fact we have to extend the configuration space by configurations representing initial and goal situations. The basic idea for modifying the algorithms to support this rests in adding 'dummy' cells - one for the initial and one for the goal situation. Cells intersecting 'dummy' cells can be considered as adjacent (to the 'dummy' cells). Defining the adjacency relation is sufficient for basic (path planning) algorithms (we do not have to change the pseudo-code).

However, for augmented algorithms it is not that simple. It may happen that if the initial speed of the entity is high, then none of cells adjacent to the initial 'dummy' cell is reachable (i.e., SLCs are violated) because the entity must perform a 'sharp' Turn Maneuver (with a small turn radius). In this case all (augmented) algorithms terminate without finding any solution. For (augmented) Theta* and Lazy Theta* there is a way to handle this sort of issue. From the initial direction and initial (minimum) speed it is possible to estimate which cell may be reachable from the initial 'dummy' cell. In fact the expansion of the initial node ($s_{start}$) might not refer to nodes positioned in adjacent cells but to nodes positioned in cells a 'reasonable' distance from the starting position (following the initial direction). A 'reasonable' distance can be determined, for instance, as a distance needed by an entity to slow-down to fulfill SLC for Turn Maneuver in angle $\pi/6$. While $s_{start}$ is being expanded, then the $ComputeCost$ procedure (Theta*, see Algorithm 7) is modified in such a way that if the test in Line 2 fails ($s_{start}$ is used instead of $parent(s)$), then we proceed to the next node (Line 12 of Algorithm 1). Analogously, the $SetNode$ procedure (Lazy Theta*, see Algorithm 8) is modified, i.e., if the test (Line 2) succeeds, then we select another node from $open$ (Line 5 of Algorithm 1).

In every iteration, when a node $s$ is selected from $open$, reachability of $s_{goal}$ is tested. That is, performing $LOS(s, s_{goal})$ followed by *verify*$(s, s_{goal})$. If both the checks succeed, then we can directly connect $s$ and $s_{goal}$, otherwise we cannot. If we optimize for distance and $spd(s_{goal})$ is not disjoint with the goal (interval of) speed values, then we can directly proceed to the solution extraction phase. If we optimize for duration, we add $s_{goal}$ to $open$ (or replace the old one if the cost is smaller) and continue (Line 5 of Algorithm 1). Therefore, the solution extraction phase begins after $s_{goal}$ is selected from $open$. Note that the *verify* procedure must also be modified for this case. The curve, which is generated (Line 2 of Algorithm 5), consists of three atomic curve segments in the following order: Turn Maneuver for changing direction towards the goal 'dummy' cell, Straight Maneuver for moving towards the goal 'dummy' cell and Turn Maneuver for changing direction towards the goal configuration. The last maneuver is omitted if direction is not specified in the goal situation.

### 7.6 Example

Figures 5,6 depict sample trajectories in the 2D environment represented by hexagonal grids where obstacles are represented by geometric objects such as circles or polygons. Trajectories are divided into atomic curve segments (Turn and Straight Maneuvers) with corresponding intervals of speed values. This shows how the intervals are developing through the trajectories (which begins at top-left corner). Straight Maneuvers allow the speed to be adjusted uniformly. For Turn Maneuvers, speed remains constant for the whole segment.

Figures 5,6 also show both trajectories touching the obstacles. That is because of the assumption we made before that entities are represented as point masses. Of course, real autonomous vehicles are not point masses, therefore the vehicles must avoid obstacles at

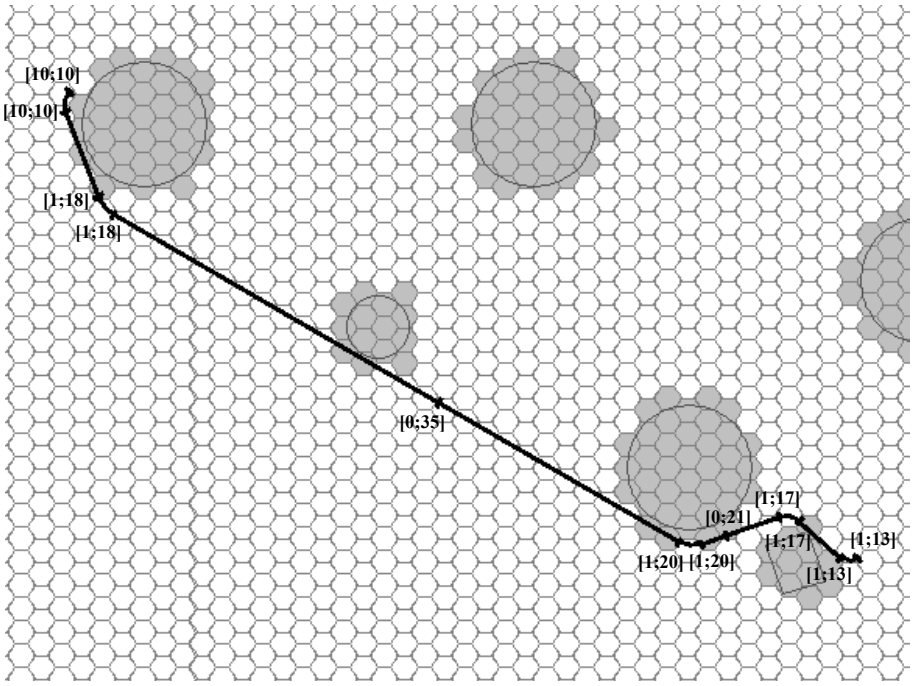


**Fig. 5** A sample trajectory, where atomic curve segments including intervals of speed values are depicted.

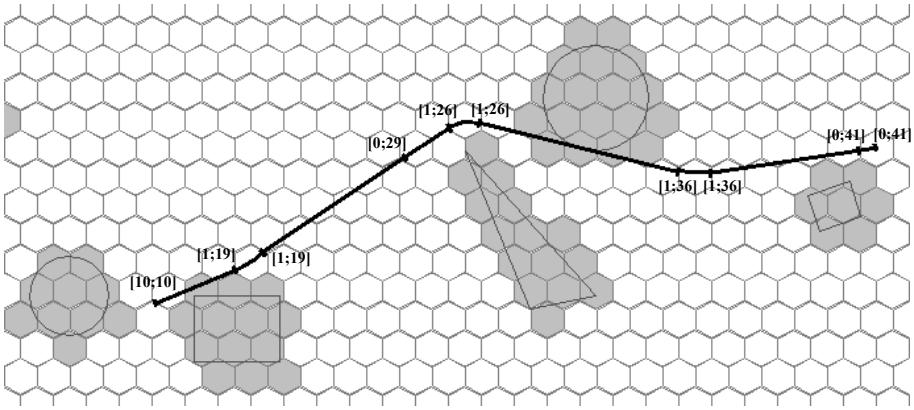


**Fig. 6** Another sample trajectory. Atomic curve segments including intervals of speed values are depicted.

some distance. To incorporate entity size or safety range (i.e., a minimum possible distance from an obstacle) the geometric objects representing obstacles must be 'inflated' (e.g. if an obstacle is represented by a circle, then we must increase its radius by the safety range).

## 8 Experimental Evaluation

In this section, we present an experimental evaluation where we compare how augmenting the A*, Theta* and Lazy Theta* algorithms affect the trajectory planning process.

### 8.1 Setup

For evaluation purposes we implemented the proposed methods for trajectory planning (augmented A*, Theta* and Lazy Theta*) in JAVA SE 6. The implementation has not been optimized for performance, therefore there might be space for improvement. All the benchmarks were performed on Core2Duo 1.86 GHz, 2 GB RAM, Windows 7. We used two scenarios for the evaluation. Firstly, a 2D scenario was built upon an area of size 1000x1000 units and is discretized into a hexagonal grid where the distance between adjacent cells is 10 units (making a grid of size approx. 100x100 cells). The Model of Dynamics for the 2D scenario, which follows the general rules described in Section 7.3, has the following properties (speed values are provided in units per second, acceleration values are provided in units per second squared): maximum speed is 50, maximum acceleration is 2.0, minimum acceleration (i.e, maximum deceleration) is -2.5 and maximum speed for a unit Turn Maneuver ($v_{tunit}$) is 5.0. Secondly, a 3D scenario was built upon an area of size 1000x1000x50 units which is discretized into a honeycomb-like grid where the distance between adjacent cells is 10 units and between adjacent 'flight levels' is 5 units (making a grid of size approx. 100x100x10 cells). The Model of Dynamics for the 3D scenario, which follows the general rules described in Section 7.3 (similarly to the 2D case), has the following properties: maximum speed is 100, maximum acceleration is 7.0, minimum acceleration (e.g, maximum deceleration) is -8.5, maximum speed for a unit Turn Maneuver ($v_{tunit}$) is 10.0 and minimum pitch radius (slope change) is 20 units.

We randomly generated 100 way-points and then we plan trajectories between each pair (giving us up to 4950 planning problems – some of them are (trivially) unsolvable, because some points lie in the blocked cells). We have three 'maps' ($\sim 10\%$, $\sim 20\%$ and $\sim 30\%$ cells blocked). We also randomly generated initial and goal direction and initial speed and maximum goal speed for every point. For every planning problem we set a timeout of 5s (5000ms). Note that we have only one set of points for all scenarios (in 2D we just omit the $z$-coordinate) and for all 'maps'.

### 8.2 Basic vs. Augmented Methods

Table 1 shows the performance of the basic (pure path planning) and augmented A*, Theta* and Lazy Theta* algorithms. The basic versions are supplemented by a post-processing step which assigns dynamics to 'paths' (spatial curves) - outputs of the (basic) A*-like algorithms. For augmented A* we considered both lite and full versions. For Theta* and Lazy Theta* we considered only lite versions because the full ones are (apart from a few exceptions) unsolvable in reasonable time. Initial configurations are specified by a (cell) position and speed value and goal configurations are specified by a (cell) position and interval of speed values (initial or goal directions are not specified).

Columns are named in a self-explanatory way but some items should be clarified. Nodes are counted in such a way that we count every node which is about to be inserted into $open$. Distance stands for a trajectory length. Duration stands for a time the entity needs to pass

| | Solved | Timeout exceeded | Running Time (ms) | | Nodes | | Distance | | Duration | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | avg | vs bas | avg | vs bas | avg | vs bas | avg | vs bas |
| $\sim 10\%$ cells blocked | | | | | | | | | | |
| A* (basic) | 3507 | 0 | 45 | | 5115 | | 525.8 | | 30.9 | |
| A* (lite) | 3917 | 0 | 46 | 1.04 | 4834 | 0.98 | 532.7 | 1.000 | 30.9 | 1.000 |
| A* (full) | 2589 | 1416 | 1276 | 63.00 | 94677 | 38.4 | 393.8 | 0.998 | 25.5 | 1.053 |
| Theta* (basic) | 3947 | 0 | 18 | | 1344 | | 515.1 | | 22.7 | |
| Theta* (lite) | 3957 | 0 | 19 | 1.07 | 1342 | 1.00 | 514.6 | 1.000 | 22.7 | 1.000 |
| Lazy Theta* (basic) | 3947 | 0 | 10 | | 1385 | | 515.1 | | 22.7 | |
| Lazy Theta* (lite) | 3957 | 0 | 10 | 1.02 | 1383 | 1.00 | 514.7 | 1.000 | 22.7 | 1.000 |
| $\sim 20\%$ cells blocked | | | | | | | | | | |
| A* (basic) | 2970 | 0 | 43 | | 4843 | | 532.6 | | 31.3 | |
| A* (lite) | 3337 | 0 | 44 | 1.03 | 4592 | 0.98 | 543.1 | 1.000 | 31.4 | 1.000 |
| A* (full) | 2197 | 1231 | 1333 | 65.91 | 96289 | 41.45 | 395.6 | 0.998 | 25.9 | 1.055 |
| Theta* (basic) | 3316 | 0 | 29 | | 1713 | | 521.8 | | 23.9 | |
| Theta* (lite) | 3361 | 0 | 31 | 1.08 | 1710 | 1.00 | 522.6 | 1.000 | 24.0 | 1.000 |
| Lazy Theta* (basic) | 3316 | 0 | 15 | | 1746 | | 521.9 | | 24.0 | |
| Lazy Theta* (lite) | 3361 | 0 | 15 | 0.99 | 1745 | 1.00 | 522.7 | 1.000 | 24.1 | 1.000 |
| $\sim 30\%$ cells blocked | | | | | | | | | | |
| A* (basic) | 2448 | 0 | 40 | | 4531 | | 553.5 | | 32.3 | |
| A* (lite) | 2805 | 0 | 41 | 1.06 | 4321 | 0.98 | 568.0 | 1.000 | 32.5 | 1.000 |
| A* (full) | 1786 | 1064 | 1405 | 74.64 | 99235 | 44.53 | 415.5 | 0.997 | 27.1 | 1.062 |
| Theta* (basic) | 2746 | 0 | 44 | | 2077 | | 539.1 | | 25.4 | |
| Theta* (lite) | 2815 | 0 | 46 | 1.06 | 2078 | 1.00 | 540.6 | 1.000 | 25.5 | 1.000 |
| Lazy Theta* (basic) | 2746 | 0 | 20 | | 2106 | | 539.3 | | 25.5 | |
| Lazy Theta* (lite) | 2815 | 0 | 20 | 1.00 | 2110 | 1.00 | 540.8 | 1.000 | 25.6 | 1.000 |

**Table 1** The performance of basic and augmented A*-like algorithms in the 2D scenario.

through the trajectory, i.e., a span of the time interval on which the trajectory is defined. To compare basic and augmented A*-like algorithms (columns 'vs bas') we used the following normalization. Let $SP$ be a set of problems solvable (within timeout) by both an augmented and the corresponding basic algorithm. Let $x_{bas_i}$ (resp. $x_{aug_i}$) be a quantity (such as running time, nodes etc.) connected with the $i$-th problem and the basic (resp. augmented) algorithm. Then, the normalization is:

$$\frac{\sum_{i \in SP} x_{aug_i}}{\sum_{i \in SP} x_{bas_i}}$$

The results are discussed in detail later.

## 8.3 Distance vs. Duration Optimization

Tables 2, 3 show the performance of the augmented Theta* and Lazy Theta* algorithms. We compare how these algorithms behave if we optimize either for trajectory distance or duration. Initial configurations are specified by a position (not necessarily in the cell center), direction and speed value and goal configurations are specified by a position (not necessarily in the cell center), direction and interval of speed values. In contrast to the previous case the algorithms are modified according to Section 7.5 (for instance, we test reachability of the goal every time we select a node for expansion).

The tables are organized in a similar way to the previous case but instead of comparing to the basic versions we are comparing to distance optimization. The normalization formula is constructed analogously to the previous case.

Results are also later discussed in detail.

| | Solved | Timeout exceeded | Running Time (ms) | | Nodes | | Distance | | Duration | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | avg | vs dist | avg | vs dist | avg | vs dist | avg | vs dist |
| $\sim 10\%$ cells blocked | | | | | | | | | | |
| Theta* (dist) | 3847 | 0 | 16 | | 504 | | 533.6 | | 31.1 | |
| Theta* (dur) | 3847 | 0 | 1002 | 61.73 | 24336 | 48.29 | 541.6 | 1.01 | 25.9 | 0.83 |
| Lazy Theta* (dist) | 3832 | 0 | 7 | | 523 | | 534.6 | | 32.5 | |
| Lazy Theta* (dur) | 3832 | 0 | 700 | 102.25 | 26372 | 50.39 | 544.2 | 1.02 | 26.2 | 0.81 |
| $\sim 20\%$ cells blocked | | | | | | | | | | |
| Theta* (dist) | 3182 | 0 | 38 | | 951 | | 542.3 | | 32.2 | |
| Theta* (dur) | 3182 | 0 | 1014 | 27.01 | 21640 | 22.75 | 551.6 | 1.02 | 26.7 | 0.83 |
| Lazy Theta* (dist) | 3169 | 0 | 15 | | 998 | | 545.6 | | 33.5 | |
| Lazy Theta* (dur) | 3169 | 0 | 687 | 46.04 | 24466 | 24.52 | 557.3 | 1.02 | 27.3 | 0.81 |
| $\sim 30\%$ cells blocked | | | | | | | | | | |
| Theta* (dist) | 2596 | 0 | 62 | | 1461 | | 560.5 | | 33.1 | |
| Theta* (dur) | 2596 | 0 | 918 | 14.91 | 18613 | 12.74 | 570.5 | 1.02 | 28.0 | 0.85 |
| Lazy Theta* (dist) | 2583 | 0 | 25 | | 1505 | | 563.8 | | 34.3 | |
| Lazy Theta* (dur) | 2583 | 0 | 659 | 26.78 | 21993 | 14.61 | 577.5 | 1.02 | 28.8 | 0.84 |

**Table 2** The performance of augmented Theta* and Lazy Theta* optimized for either distance or duration in the 2D scenario.

| | Solved | Timeout exceeded | Running Time (ms) | | Nodes | | Distance | | Duration | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | avg | vs dist | avg | vs dist | avg | vs dist | avg | vs dist |
| $\sim 10\%$ cells blocked | | | | | | | | | | |
| Theta* (dist) | 3809 | 0 | 170 | | 4123 | | 513.8 | | 17.3 | |
| Theta* (dur) | 491 | 3318 | 2661 | 235.68 | 59731 | 226.74 | 186.1 | 1.05 | 8.2 | 0.85 |
| Lazy Theta* (dist) | 3809 | 0 | 66 | | 4215 | | 514.3 | | 17.8 | |
| Lazy Theta* (dur) | 759 | 3050 | 2842 | 400.51 | 85899 | 202.20 | 220.6 | 1.04 | 9.0 | 0.86 |
| $\sim 20\%$ cells blocked | | | | | | | | | | |
| Theta* (dist) | 3250 | 11 | 447 | | 9111 | | 518.1 | | 18.0 | |
| Theta* (dur) | 415 | 2846 | 2792 | 135.62 | 54850 | 136.45 | 182.2 | 1.05 | 8.1 | 0.84 |
| Lazy Theta* (dist) | 3261 | 0 | 169 | | 9652 | | 520.1 | | 18.5 | |
| Lazy Theta* (dur) | 693 | 2568 | 2550 | 234.61 | 83643 | 127.42 | 222.3 | 1.05 | 9.1 | 0.86 |
| $\sim 30\%$ cells blocked | | | | | | | | | | |
| Theta* (dist) | 2725 | 23 | 740 | | 14419 | | 534.8 | | 18.6 | |
| Theta* (dur) | 367 | 2381 | 2779 | 61.23 | 48203 | 62.84 | 190.1 | 1.05 | 8.4 | 0.84 |
| Lazy Theta* (dist) | 2748 | 0 | 296 | | 15483 | | 538.9 | | 19.2 | |
| Lazy Theta* (dur) | 628 | 2120 | 2554 | 84.88 | 77706 | 47.12 | 244.0 | 1.05 | 9.7 | 0.84 |

**Table 3** The performance of augmented Theta* and Lazy Theta* optimized for either distance or duration in the 3D scenario.

### 8.4 Discussion

The comparison of the basic and augmented A*-like algorithms (Table 1) shows the following. In several cases the basic algorithms (dynamics is added in a post-processing step) were not able to find a solution trajectory although the augmented algorithms were. It happened more often in the A* case, that is because the smallest turn angle is $\pi/6$ (in a hexagonal grid with 'diagonal' moves) thus the maximum speed for this Turn maneuver is quite low. There is a correlation between the number of obstacles and the number of 'unsolvable' problems by the basic versions. We have observed that the main reason why problems are 'unsolvable' by the basic versions is a high initial speed and an early 'sharp' Turn Maneuver. This means that the entity is not able to slow-down to perform the Turn maneuver. Note that initial and goal directions are not specified in this case, which helps the basic versions. Taking a closer look at the results (Table 1) we can see that the hypothesis formulated in Section 6.4 is justified. The number of considered nodes was almost the same for the basic and lite versions, in the A* case it was about $2\%$ less in the lite versions. Running time of the lite versions was slightly higher (up to $8\%$) than running time of the corresponding basic versions. This is caused by higher overheads connected with computation of intervals of speed values and de-

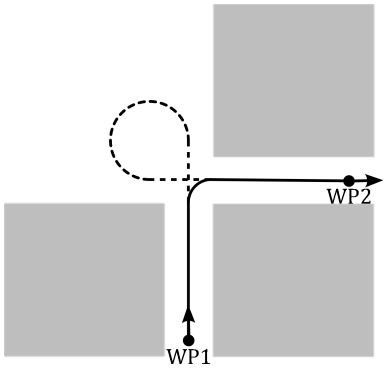


**Fig. 7** Illustration of two possible trajectories between WP1 and WP2. Dash line illustrates a situation when minimal possible turn radius is too large and the trajectory forms an 'ear'.

termining SLCs. It is worth mentioning that the running time of the lite version of augmented Lazy Theta* is almost the same as the corresponding (basic) Lazy Theta*. We believe that postponing these overheads along with LOS checks which results in a significant decrease of these overheads (in fact we have to determine SLCs and compute $spd(s)$ only for nodes that are to be inserted into $close$ rather than into $open$). Full versions of augmented A* have, as we expected, about 65-75 times higher running time than the corresponding basic versions. This is also in a correlation with the number of considered nodes which is about 40 times higher. We can see that in our case (hexagonal grid with 'diagonal' moves) the number of directions is 12. We can estimate the average number of different intervals of speed values for every considered node ($c$, see Section 6.4) to be 3-4. However, this estimation is inaccurate and very dependent on a particular model of dynamics. In Theta* and Lazy Theta* case the number of directions is very high thus running time of the full versions might be extremely high (see [7]). Trajectories found by the lite versions do not differ much from trajectories found by the basic versions since the differences between trajectory lengths and duration is marginal (fractions of promile). Comparing the full versions of augmented A* with the corresponding basic versions we can see that trajectory lengths are slightly shorter (2-3 promiles), however, durations are about $5\%$ higher for the full versions. This is due to a different strategy for distinguishing nodes which often results in trajectories that contain more Turn maneuvers. Even though distances (lengths of trajectories) are not (or very slightly) affected, durations are longer because Turn maneuvers can be performed only at reduced speed.

The lite versions were able to solve more problems than the basic versions at only a small price. However, incompleteness of the lite version is their major shortcoming. As discussed in Section 6.4 solutions might be lost because of comparing only positions of nodes, therefore we might discard some essential configurations. In some cases we have to reenter a cell as illustrated in Figure 7. A dash-lined trajectory must be considered if the initial speed (in WP1) is too high, therefore we are unable to perform a 'normal' Turn maneuver. In general if we have visited such a position (cell) once, then we will not visit it again regardless of differences of other conditions (direction, interval of speed values). Encouragingly we have only observed a small number of such lite version 'unsolvable' problems. However, if such an 'unsolvable' problem occurs, then one option is to try to solve it by the full version of augmented A*, which has been proven to be complete in Section 6.4. Main disadvantage of this approach is quite low performance as well as rather

unrealistic trajectories. Another option, which is still an open problem, rests in detecting positions (cells) that must be reentered or revisited.

Comparisons of the results of using a different optimization strategy (for distance or duration) are present in Tables 2 (for the 2D scenario) and 3 (for the 3D scenario). We used the lite versions of Theta* and Lazy Theta*. Initial configurations in this case have arbitrary positions (not necessarily in cell centers), directions and speed values. Goal configurations in this case have also arbitrary positions, directions and intervals of speed values. Note that in this case using the basic versions is quite impractical because in many cases the first Turn maneuver, which goes from the initial configuration, has too small turn radius thus it does not follow the dynamic properties of the entity. The results showed that we are able to reduce trajectory durations (if we optimize for it) by about $15 - 18\%$ while trajectory lengths increase by about $2 - 5\%$. However, optimization for duration turned out to be significantly much slower than optimization for distance. There are several aspects influencing this. Firstly, the cost of nodes ($g$) depends on time needed to pass through a partial trajectory (from the initial to the current node). Obviously, it is possible for longer (partial) trajectories to have lower cost than shorter ones. In our case the cost rises in correlation with the number and 'sharpness' of Turn maneuvers which means that 'straight' trajectories are preferred. It is also obvious that the number of considered nodes rises when the (partial) trajectories become longer. It gives us an insight why the number of considered nodes must be higher. The heuristic function ($h$), on the other hand, is directly dependent on a Cartesian distance from the current (set of) configurations to the goal configurations (see Section 7.4). Thus, the heuristic does not affect the number of considered nodes. Secondly, if we optimize for distance we can 'jump' directly to the goal when it becomes reachable (see Section 7.5). On the other hand, if we optimize for duration and even if the goal is reachable we cannot directly 'jump' to it. The results show that if the number of obstacles rises then the difference between considered nodes (and running time as well) decreases. This is because having more obstacles decreases a chance of 'jumping' towards the goal. The third aspect affects only running time. It is obvious that computing the cost ($g$) is more time consuming because we have to produce a partial trajectory (i.e., assign velocity to the sequence of curve segments) in every iteration. Addressing these aspects offers potential improvements of the planning process (optimizing for duration). The first and most important aspect can be addressed by developing more sophisticated (admissible) heuristics that avoid the exploration of a large number of unpromising nodes. This is also related to the second aspect because with more sophisticated heuristics we will be able to reveal earlier whether 'jumping' towards the goal is feasible with respect to duration optimization.

Comparing the results of our methods in 2D and 3D scenarios (using a different model of dynamics) showed that there is a correlation between the number of considered nodes and grid size in the distance optimization case. In the duration optimization case the number of considered nodes is significantly higher. Therefore, we can see that particular model of dynamics has a much greater effect on planning optimized for duration.

## 9 Conclusions

We used the concept of trajectory planning which is designed as a modular architecture allowing independent and easy implementation of spatial and dynamic parts. Because spatial aspects can be addressed by path planning methods, which are very popular, we showed, in this paper, how these path planning methods, namely A*, Theta* and Lazy Theta*, can be augmented to support such a concept of trajectory planning. In contrast to some existing

approaches, which after finding a 'path' (a curve) by path planning methods add dynamics (speed and time) to obtain a trajectory in a post-processing step, our approach handles dynamic properties (such as SLCs) during the search. We presented lite and full versions of the augmented algorithms which differ in how they distinguish nodes. The lite version of augmented Lazy Theta* especially achieved very good and promising results. Overall, the lite versions were able to solve more problems than the basic versions (non-augmented path planning methods) at the cost of a small increase in running time. Moreover, we formally proved the completeness of the full version of augmented A*. However, experiments showed that the full version is significantly slower (as we expected) than the corresponding lite (or basic) version.

In this paper we also presented how we can extend augmented Theta* and Lazy Theta* algorithms to plan trajectories from and to arbitrary positions, directions and (intervals of) speed value(s). We investigated how this approach behaves when we optimize for distance (length of a solution trajectory) or duration (a time span a solution trajectory is defined on). Performance of planning optimized for duration was significantly lower than performance of planning optimized for distance. We identified aspects which cause this drop of performance. Addressing these aspects will be a part of our future work.

Future research should consist of proposing more sophisticated heuristics for trajectory planning optimized for duration (or, for instance, energy consumption). Such a heuristic should be able to reduce the number of considered nodes, which will address some of the aspects we discussed earlier in this paper. Another open problem we discussed in this paper concerns incompleteness of the lite versions of augmented (A*-like) algorithms. The problem rests in detecting positions (cells) that may be reentered or revisited, because if we have visited such a position (cell) once, then we cannot visit it again regardless of differences of conditions (direction, interval of speed values).

In summary, the trajectory planning concept introduced here has shown its usefulness despite its very basic implementation of Trajectory Representation and Model of Dynamics components. We believe that results of our work will be applied in simulations of complex autonomous systems which can be used as feasibility studies for real-world applications.

## References


1. Ahmadzadeh, A., Motee, N., Jadbabaie, A., Pappas, G.J.: Multi-vehicle path planning in dynamically changing environments. In: Proceedings of ICRA, pp. 2449–2454 (2009)
2. Barraquand, J., Latombe, J.C.: Nonholonomic multibody mobile robots: Controllability and motion planning in the presence of obstacles. Algorithmica **10**(2-4), 121–155 (1993)
3. Botea, A., Müller, M., Schaeffer, J.: Near optimal hierarchical path-finding. Journal of Game Development **1**, 7–28 (2004)
4. Bresenham, J.: Algorithm for computer control of a digital plotter. IBM Systems Journal **4**(1), 25–30 (1965)
5. Cheng, P., LaValle, S.M.: Resolution complete rapidly-exploring random trees. In: Proceedings of ICRA, pp. 267–272 (2002)
6. Choset, H., Lynch, K.M., Hutchinson, S., Kantor, G.A., Burgard, W., Kavraki, L.E., Thrun, S.: Principles of Robot Motion: Theory, Algorithms, and Implementations. MIT Press, Cambridge, MA (2005)
7. Chrpa, L.: Trajectory planning on grids: Considering speed limit constraints. In: Proceedings of SCAI, pp. 60–69. IOS press (2011)
8. Chrpa, L., Komenda, A.: Smoothed hex-grid trajectory planning using helicopter dynamics. In: Proceedings of International Conference on Agents and Artificial Intelligence (ICAART), vol. 1, pp. 629–632 (2011)
9. Daniel, K., Nash, A., Koenig, S., Felner, A.: Theta*: Any-angle path planning on grids. Journal of Artificial Intelligence Research (JAIR) **39**, 533–579 (2010)
10. Demyen, D., Buro, M.: Efficient triangulation-based pathfinding. In: Proceedings of AAAI (2006)

11. Ferguson, D., Howard, T.M., Likhachev, M.: Motion planning in urban environments. In: The DARPA Urban Challenge, pp. 61–89 (2009)
12. Ferguson, D., Stentz, A.: Using interpolation to improve path planning: The field d$^*$ algorithm. Journal of Field Robotics **23**(2), 79–101 (2006)
13. Ghallab, M., Nau, D., Traverso, P.: Automated planning, theory and practice. Morgan Kaufmann Publishers (2004)
14. Hart, P., Nilsson, N., Raphael, B.: A formal basis for the heuristic determination of minimum cost paths. IEEE Transactions on Systems Science and Cybernetics **4**(2), 100–107 (1968)
15. Howard, T.M., Kelly, A.: Optimal rough terrain trajectory generation for wheeled mobile robots. International Journal of Robotic Research **26**(2), 141–166 (2007)
16. Karaman, S., Walter, M.R., Perez, A., Frazzoli, E., Teller, S.J.: Anytime motion planning using the rrt*. In: Proceedings of ICRA, pp. 1478–1483 (2011)
17. Kavraki, L.E., Svestka, P., Kavraki, L.E., Latombe, J.C., Overmars, M.H.: Probabilistic roadmaps for path planning in high-dimensional configuration spaces. IEEE Transactions on Robotics and Automation **12**, 566–580 (1996)
18. Latombe, J.C.: Robot Motion Planning. Kluwer Academic Publishers, Norwell, MA, USA (1991)
19. Likhachev, M., Ferguson, D., Gordon, G.J., Stentz, A., Thrun, S.: Anytime search in dynamic graphs. Artificial Intelligence **172**(14), 1613–1643 (2008)
20. Lozano-Pérez, T., Wesley, M.A.: An algorithm for planning collision-free paths among polyhedral obstacles. Commununications of the ACM **22**(10), 560–570 (1979)
21. Mehlhorn, K.: Data Structures and Algorithms 2: Graph Algorithms and NP-Completeness. Springer-Verlag (1984)
22. Narayanan, V., Phillips, M., Likhachev, M.: Anytime safe interval path planning for dynamic environments. In: Proceedings of IROS, pp. 4708–4715 (2012)
23. Nash, A., Daniel, K., Koenig, S., Felner, A.: Theta*: Any-angle path planning on grids. In: Proceedings of AAAI, pp. 1177–1183 (2007)
24. Nash, A., Koenig, S., Likhachev, M.: Incremental phi*: Incremental any-angle path planning on grids. In: Proceedings of IJCAI, pp. 1824–1830 (2009)
25. Nash, A., Koenig, S., Tovey, C.A.: Lazy theta*: Any-angle path planning and path length analysis in 3d. In: AAAI (2010)
26. O'Rourke, J.: Computational geometry in C. Cambridge University Press (1998)
27. Phillips, M., Likhachev, M.: Sipp: Safe interval path planning for dynamic environments. In: Proceedings of ICRA, pp. 5628–5635 (2011)
28. Pivtoraiko, M., Knepper, R.A., Kelly, A.: Differentially constrained mobile robot motion planning in state lattices. Journal of Field Robotics **26**(3), 308–333 (2009)
29. Scheuer, A., Fraichard, T.: Continuous-curvature path planning for car-like vehicles. In: Proceedings of the IEEE-RSJ International Conference on Intelligent Robots and Systems, pp. 997–1003 (1998)
30. Šišlák, D., Pěchouček, M., Volf, P., Pavlíček, D., Samek, J., Mařík, V., Losiewicz, P.: Defense Industry Applications of Autonomous Agents and Multi-Agent Systems, chap. AGENTFLY: Towards Multi-Agent Technology in Free Flight Air Traffic Control, pp. 73–97. Birkhauser Verlag (2008)
31. Wang, W., Xu, X., Li, Y., Song, J., He, H.: Triple rrts: An effective method for path planning in narrow passages. Advanced Robotics **24**(7), 943–962 (2010)
32. Wzorek, M., Doherty, P.: Reconfigurable path planning for an autonomous unmanned aerial vehicle. In: Proceedings of ICAPS, pp. 438–441 (2006)
33. Yap, P.: Grid-based path-finding. In: Proceedings of Canadian Conference on AI, pp. 44–55 (2002)