

Article

Late Line-of-Sight Check and Prioritized Trees for Path Planning

Changwu Zhang ^{1,2,*} , Yuchen Tang ³, Li Zhou ^{1,*} and Hengzhu Liu ¹¹ College of Computer, National University of Defense Technology, Changsha 410073, China² State Key Laboratory of High Performance Computing, National University of Defense Technology, Changsha 410073, China³ Faculty of Engineering, The University of Hong Kong, Hong Kong 999077, China

* Correspondence: woody_chang@yeah.net (C.Z.); zhouli06@nudt.edu.cn (L.Z.)

Received: 17 July 2019; Accepted: 16 August 2019; Published: 21 August 2019



Abstract: How to generate a safe and high-quality path for a moving robot is a classical issue. The relative solution is also suitable for computer games and animation. To make the problem simpler, it is common to discretize the continuous planning space into grids, with blocked cells to represent obstacles. However, grid-based path-finding algorithms usually cannot find the truly shortest path because the extending paths are constrained to grid edges. Meanwhile, Line-of-Sight Check (LoS-Check) is an efficient operation to find the shorter any-angle path by relaxing the constraint of grid edges, which has been successfully used in Theta*. Through reducing the number of LoS-Check operations in Theta*, a variant version called LazyTheta* speeds up the planning especially in the 3D environment. We propose Late LoS-Check A* (LLA*) to further reduce the LoS-Check amount. It uses the structure of the prioritized trees to partially update the gvalues of different successors that share the same parent (i.e., the parent of the current vertex waiting to be extended). The sufficient experiments on various benchmark maps show that LLA* costs less execution time than Lazy Theta* while generating shorter paths. If we just delay the LoS-Check, the path planned by LLA* will hardly be shorter than that of Lazy Theta*. Therefore, the key of LLA* is the discriminatory strategy, and we empirically explain the reason why both the path length and execution time of LLA* are shorter than those of Lazy Theta*.

Keywords: path planning; line-of-sight; lazy evaluation; robotic planning; path finding

1. Introduction

Path planning plays an important role in robotic planning, and it has a long history in the field of artificial intelligence as well. The goal of path planning is to find a feasible path between two given points, and the best result is finding the shortest path. There are many different kinds of strategies to discretize the continuous planning environment [1]. Two representative categories are graph-based planning and sampling-based planning.

Graph-based planning is usually based on grids [2,3] or navigation meshes [4] and then uses dynamic programming [5] (e.g., the Dijkstra algorithm [6] and A* algorithm [7]) to search for a path. Grids save the whole information of a map, which is suitable for a dynamic environment. Grid-based path planning algorithms are commonly used in computer games and robotics, and they discretize the continuous environment into discrete grids with blocked cells to represent obstacles. Navigation meshes are constructed on the basis of a collection of convex polygons in the free space, which require smaller memory and have lower density. However, when the environment changes frequently, navigation meshes need to be reconstructed correspondingly.

Sampling-based planning discretizes the continuous planning domain by randomly sampling the state space, avoiding constructing maps for the entire environment [8]. Therefore, sampling-based planners can generate a feasible path fast, especially in a high-dimensional space. However, the biggest problem of these methods is that they almost cannot find the optimal solution easily, such as RRT [9] and PRM [10]. RRT* [11] is a variant of RRT, and it employs many local rewiring operations to optimize the path. RRT* could asymptotically obtain the optimal solution as the number of iterations increases, but the efficiency is not quite ideal. Sampling-based planners have good performance in robotic manipulation [12] and mobile robots [13].

There are many other kinds of path planning algorithms as well. Artificial Potential Field (APF) [14] is a kind of real-time planner, which divides the planning environment into different fields of forces. The goal position is considered as an attractive field, which guides the agent to move ahead. Meanwhile, the obstacles are described as repulsive fields, which forbid the agent from getting close. The motivation of APF is very easy to understand, and it has many variants. The most important problem of APF is how to avoid being trapped in local minima.

Reinforcement Learning (RL) [15] is also a kind of popular technique for path planning. RL-based planning uses the Markov Decision Process (MDP) [16,17] to construct the model and introduces a reward mechanism. When the agent reaches a good place, it will return a reward. Otherwise, it will get a punishment. The goal of RL is to get the biggest rewards. Value iteration and policy iteration are two main methods to solve MDP [18]. Q-learning [19] is a simple method for agents to learn to act optimally, and it can converge to the optimum action values with probability one because every state in the domain could be visited. Reinforcement learning often uses deep neural networks for feature extraction, which is often called Deep Reinforcement Learning (DRL). Deep Q-Network (DQN) [20] is a representative of the DRL model, which only uses pixels and game scores to play the Atari 2600 games. The Value Iteration Network (VIN) [21] uses a convolutional neural network to approximate the value iteration, and it is shown to generalize better to the new planning domain. The Value Prediction Network (VPN) [22] combines model-free and model-based RL methods, outperforming DQN on several Atari games. These DRL-based methods have been successfully applied in path planning, but a common problem is that they cannot deal with the large scale of maps quickly.

How to generate a safe and high-quality path for the robots or animated characters is a popular issue in the path planning problems. A classical scene is moving a 3D model of a piano from one corner to another corner in a room, which is shown in Figure 1. The sampling-based RRT algorithm has been successfully used in the robotic planning and character animation [23]. Meanwhile, the bidirectional version of RRT, which is called RRT-Connect [24], was also applied to generate collision-free motions for the robots. Similarly, the A* algorithm was used to generate realistic motions for moving robots or animated characters, and these motions are abstracted as high-level behaviors [25]. Precomputed Search Trees (PST) [26] is an efficient approach to synthesize motions for animated characters navigating in given environments, and it is faster than A*. Moreover, the crowd animation planning is a complex problem with many parameters to be tuned, and there are several good methods in this domain [27,28].



Figure 1. Moving a piano from one corner to another corner using the LLA* algorithm (top view). The left figure shows the start place and the goal place of the piano (from the left corner to the right corner). The right figure shows the composite moving trajectory of the piano without touching the tables and chairs.

1.1. Any-Angle Path Planning

It is popular to use the A* [7] algorithm for pathfinding on grids or navigation meshes. A* can usually find the optimal solution along with grid edges, but it is not the real shortest path because the paths are constrained to grid edges [2]. To relax this kind of constraint and obtain a more ideal and shorter path, many types of research attempts have been made. Related methods are called any-angle path planning, meaning that the path will not be constrained to grid edges. Line-of-Sight Check (LoS-Check) is an efficient method to relax the constraint of grid edges via checking whether there is a straight line between two non-adjacent vertices. If the straight path does not traverse the obstacles, the updated path will be shorter according to the triangle inequality. LoS-Check can be efficiently implemented by the line-drawing algorithm [29].

Any-angle path planning can be divided into three main categories: post-processing, online-processing, and preprocessing.

Post-processing is a method that obtains the grid-optimal path firstly and then uses the shortcut to replace the old path between those points, which satisfies the line-of-sight condition [30]. Although this kind of LoS-Check can smooth the planned path, actually it can hardly result in the real shortest solution.

More and more any-angle path planning algorithms are concentrating on online-processing. Field D* [31] uses linear interpolation on grid edges to relax the edge constraint, but it is not optimal. Theta* [2] performs LoS-Check between the parent of the current vertex and every successor of the current vertex, showing that the planned path is close to the shortest. Lazy Theta* [32] is a variant of Theta*, which adopts the lazy evaluation strategy to reduce the amount of LoS-Check, and it shows good performance in a 3D environment. Lazy Theta* has been used in the path planning of a multirotor UAV [33]. There is also an optimal any-angle pathfinding algorithm called Anya [34], which can always return an optimal path without any preprocessing or memory overheads. Anya is an optimal and online any-angle path planning algorithm, but it will cost more time on the “interval” search. Meanwhile, Uras and Koenig [35] showed that Anya is slower than A* and Theta* on average.

Preprocessing is often used to accelerate the planning speed, which usually needs some computation before the main planning, aiming to collect more informed information about the given environment. Block A* [36] employs a preprocessed database, which contains the distances between local points. Subgoal graphs [37] are preprocessed from grids by placing subgoals at the corners of obstacles and adding edges between the necessary ones. PolyAnya [4] is a variant of Anya based on the navigation mesh, but it may cost more time on constructing the navigation meshes. These methods can speed up any-angle path planning.

Sometimes, there is no need to get the strictly shortest path for Real-Time strategy (RTS) games, and the most important goal is to get a relatively short path as quickly as possible. Considering that

the environment of a game may change frequently, the online-processing any-angle path planning algorithm is a better choice. Therefore, we design our algorithms on the basis of A* and use less LoS-Check to optimize the path, aiming to get a shorter path with less time simultaneously.

1.2. Related Work and Motivation

The purpose of this paper is to design an algorithm that can generate better paths within less time, and the relevant techniques include Line-of-Sight Check (LoS-Check), the lazy evaluation strategy, discriminatory updating, and prioritized trees.

LoS-Check is a kind of path shortcutting method, which is usually performed to validate whether there is a straight connection between two non-adjacent states. LoS-Check is commonly employed in graph-based path planning algorithms, such as Theta* [2] and Lazy Theta* [32], which have been mentioned in the prior subsection. Of course, this kind of online path optimization method is also used widely in sampling-based path planning. The RRT*-Smart [38] algorithm employs LoS-Check and intelligent sampling to accelerate the converging rate and reach a better solution at a faster speed. RRT*-gp [39] uses the LoS-Check to connect new states with their grandparents, which makes the final solution cost converge faster to the optimum. LoS-Check helps the planners generate better solutions at a faster speed, but it could still be improved. There are also many other path optimization methods similar to LoS-Check [40–42].

The lazy evaluation strategy has been successfully used to reduce the computation on the collision detection. Collision detection is the bottleneck of the path planning problems, and it will make the solving process become slow, especially in high-dimensional spaces. In fact, some collision detection is not necessary in the planning process, which could be delayed or omitted. Lazy PRM [43] is a variant of PRM [10], which delays the collision detection until it is necessary, and it shows faster speed than PRM. Lazy Theta* [32] is the variant of Theta* [2], which shares a similar strategy as Lazy PRM, and it shows faster speed, especially in 3D grid maps. The Fast Marching Trees (FMT*) [44] algorithm performs the lazy strategy through dynamic programming recursion, which only detects the collision of the state connection that could improve the current solution. Lower Bound Tree-RRT (LBT-RRT) [45] employs the lower bound value to delay the collision detection, which runs faster and generates shorter paths when compared with RRT*. Batch Informed Trees (BIT*) [46] uses a candidate edge queue as a buffer to delay the collision detection of the edges.

Motivation: LoS-Check is a kind of long-distance collision detection, which enlarges the range of path optimization. However, it will cost much time in high-dimensional spaces. Meanwhile, the lazy evaluation strategy is often used to reduce the collision detection, which could save time. In this paper, we combine LoS-Check with the lazy evaluation strategy on the basis of the A* algorithm and construct a prioritized tree to choose which state should be performed with LoS-Check.

Moreover, the A* algorithm has many variants, which could use LoS-Check as well. The D* [47] algorithm is the Dynamic version of the A* algorithm and plans the path from the goal state to the start state. D* is applicable in unknown, partially-known, and dynamic environments. The focused D* [48] algorithm is an extension to the D* algorithm, which focuses the re-planning process to decrease the total execution time when the environment changes. Lifelong Planning A* (LPA*) [49] is an incremental search algorithm, and it does not need to be restarted when the environment changes. Thus, LPA* is suitable for a dynamic environment as well. The D* Lite [50] algorithm combines the dynamic feature of the D* algorithm with the incremental search feature of the LPA* algorithm, showing better performance in practice. LoS-Check with discriminatory updating could be employed in the planning process of these variants of A* as well. We hope this kind of discriminatory principle could give some reference and inspiration to peers in the field of robotic planning.

1.3. Contributions and Organization

Any-angle pathfinding methods usually result in shorter paths. Inspired by the delayed strategy of Lazy Theta*, we present Late LoS-Check A* (LLA*) to further delay the LoS-Check while updating

the g values of different successors with discriminatory principles. LLA* presented in this paper is based on the tree structure, which is especially useful for path planning under differential constraints. The contributions of this paper include:

- Combine LoS-Check, the late evaluation strategy, and discriminatory updating together.
- Use the prioritized trees to implement the discriminatory updating.
- Present an any-angle post-smoothing method for path planning.
- Evaluate our algorithms on some benchmark maps and integrate them into the Open Motion Planning Library (OMPL) [51].

This paper is a modified and extended version of a publication of IET Electronics Letters [52]. The original version only described a simple and short outline of the LLA* algorithm, which is based on the graph structure. In this paper, we present additional experiments and implement LLA* on the basis of OMPL with a tree structure. We also present a post-smoothing method based on LoS-Check to validate the performance of the LLA* algorithm. The final results show that LLA* runs faster and generates shorter paths than Lazy Theta*.

LLA* and Lazy Theta* both use the lazy evaluation strategy, and the biggest difference between them is that LLA* also employs the discriminatory principles. In Lazy Theta*, every state has the equivalent chance to be performed with LoS-Check, while in LLA*, LoS-Check is only performed on those states that own a lower estimate cost. It is essentially a “winner takes all” strategy. Namely, the state that has a low estimate cost will get even lower cost after LoS-Check. The discriminatory updating is a kind of greedy strategy, which could speed up LLA*. The worst case for LLA* is degrading into the A* algorithm, so it always generates shorter than or equivalent paths to A*.

Paper organization: The remainder of this paper is organized as follows. Section 2 defines some important notations, which will be used in context. Section 3 describes the mechanisms of A*, Theta*, and Lazy Theta*. The key part of this paper is given in Section 4, which explains the details of the LLA* algorithm, including the late LoS-Check operation and the discriminatory updating strategy. A kind of post-smoothing method based on the LoS-Check is introduced in Section 5. To verify the performance of LLA*, Section 6 presents the results of the sufficient experiments comparing LLA* with the other three algorithms on some benchmark maps. Lastly, we give a conclusion about our work in Section 7.

2. Definition

The n -dimensional planning environment is often continuous, and we define it as $Env \subseteq R^n$. The first step is to discretize Env into discrete grids with blocked or unblocked cells, which can be represented as $Cells = \{cell_1, cell_2 \dots cell_\infty\}$. The vertices on these grids are denoted as $V = \{v_1, v_2 \dots v_\infty\} \subseteq R^n$. Then, $v_{start} \in Env$ is the start vertex in the planning, and $v_{goal} \in Env$ is the goal vertex. If the planning domain is the 2D space ($Env \subseteq R^2$), we assume it extends in eight neighbor directions. If it is a 3D environment ($Env \subseteq R^3$), the 26-neighbor extension could be considered. An agent will take actions to reach a new vertex, and the action list is notated as $Actions$. We define the successor vertices of the current vertex as $Succ(v)$, and a vertex may have many successors. The successor vertex extension process is based on the location of the current vertex v_k , the neighbor $cell \in Cells$, and the $action \in Actions$. Then, this process is defined as $Succ(v_k) = Extend(v_k, cell, action)$. Our algorithm is based on the tree structure, and every vertex has only one parent. The parent of vertex v is notated as $Parent(v)$. The cost between two vertices is calculated upon the Euclidean distance, which is denoted as $Cost(v_m, v_n)$.

We used LoS-Check in our algorithms, which is defined as $LineofSight(v_m, v_n)$, and it will return *True* if the straight segment between v_m and v_n does not cross the blocked cells. If v_n is the successor of v_m , we can perform $Parent(v_n) \leftarrow v_m$ if $LineofSight(v_m, v_n)$ is *True*. Therefore, $Succ(v)$ and $Parent(v)$ may not be the neighbors of v after LoS-Check. The final planned path is represented as $Path = \{v_{start}, \dots, v_k, v_{k+1}, \dots, Parent(v_{goal}), v_{goal} \mid v_k = Parent(v_{k+1})\}$, and every

vertex in *Path* is the only parent of the latter one. Lastly, we define the goal of the path planning as the following representation:

$$\min \sum_{v=v_{goal}}^{v_{start}} Cost(Parent(v), v), \quad v \in Path \subseteq R^n \quad (1)$$

From Formula (1), it is not difficult to discover: in the path planning, which is based on the tree structure, a very important task is to find the best parent for every state. We also call the “vertex” as the “node” or “state” in this paper.

3. Forward Estimation and Online Optimizing

A* is the basis of many any-angle path planning algorithms, including the Theta* family. The basic A* defines the cost of node n with a synthetic value $f(n) = g(n) + h(n)$. The value $g(n)$ is the real cost from the start node to node n . The value $h(n)$ represents the estimation cost from node n to the goal node, which is often called the heuristic function. Weighted A* [53] is a variant of A* introducing the technique of multiplying the heuristic function by a weight w , which is a real value greater than or equal to one. The final cost of node n is defined as $f(n) = g(n) + w * h(n)$. We change the f value of Weighted A* (WA*) into the following form:

$$f(n) = (1 - \lambda) * g(n) + \lambda * h(n), \quad \lambda \in [0, 1] \quad (2)$$

According to Formula (2), when $\lambda = 0$, WA* will become Dijkstra [6], and when $\lambda = 1$, WA* will become the greedy best-first search [54]. When $\lambda = 0.5$, it will become A*. We control the greedy degree of WA* by changing the equilibrium factor λ . In this paper, we use the Euclidean distance as the heuristic, which satisfies both admissibility and consistency [7], and the two properties can guarantee optimality. A heuristic is admissible if $h(n) < Cost(n, goal)$, and it is consistent if $h(n) < Cost(n, n') + h(n')$, where node n' represents every successor of node n .

The A* algorithm contains two states sets, i.e., *open* and *closed*. The states in the *open* set represent that they are visited, but have not been validated to be optimal, and the states in the *closed* set represent that they have been verified to be optimal. The *open* set is a prioritized queue, and the state that has the least f value will be chosen to expand new states in each iteration. After the state expansion, this chosen state will be moved from the *open* set to the *closed* set. The A* algorithm is both resolution complete and resolution optimal because it satisfies the monotone condition that the solution cost estimate (i.e., the f value) from a parent to its child state must be monotonically increasing [34].

The contribution of A* is advising the heuristic function $h(n)$ to speed up the search, which is a kind of forward estimation about the future cost. Meanwhile, the contribution of Theta* is introducing the online LoS-Check to optimize the value $g(n)$, which reduces the real cost.

3.1. Theta*

Theta* adds the online LoS-Check into A* when the successors of the current vertex are generated. Assume the current vertex is v and its parent vertex is p . If there is the *LineofSight* between vertex p and any successor of v , set vertex p as the parent of the successor. If the *LineofSight* condition is not satisfied, maintain v as the parent of the successor. Figure 2 (left) shows the extension process of Theta*, and when the vertices $\{A2, A3, B3, C3, A4, B4\}$ are generated, there will be LoS-Check between their grandparent vertices and them.

Algorithms 1 and 2 review the specific implementation of Theta* and Lazy Theta*. The sentences in the square brackets mean that they are used in other algorithms, and the annotation gives some instruction and explanation. If the code in Line 12 of Algorithm 2 is activated, Algorithms 1 and 2 will become Theta*. Line 4 in Algorithm 1 means popping out the state with the least f value from the *open* list to act as the current state.

Algorithm 1: Theta* and Lazy Theta*

```

1   $g(v_{start}) \leftarrow 0;$      $closed \leftarrow \emptyset;$      $open \leftarrow \emptyset;$ 
2   $open.Add(v_{start}, g(v_{start}) + h(v_{start}));$ 
3  while  $open \neq \emptyset$  do
4     $current \leftarrow open.Pop\_least();$ 
5    [ $current \leftarrow UpdateParent(current);$ ]    //Lazy Theta*
6    if  $current == goal$  then
7      return  $path;$ 
8     $closed.Add(current);$ 
9    for each  $v \in Succ(current)$  do
10     if  $v \notin closed$  then
11       if  $v \notin open$  then
12          $g(v) \leftarrow \infty;$ 
13          $Parent(v) \leftarrow NULL;$ 
14          $UpdateNode(v, current);$ 

```

Algorithm 2: Relative procedures

```

1  UpdateNode( $v, current$ )
2     $g\_old \leftarrow g(v);$ 
3     $CompareCost(v, current);$ 
4    if  $g(v) < g\_old$  then
5      if  $v \in open$  then
6         $open.Delete\_old(v);$ 
7         $open.Add(v, g(v) + h(v));$ 
8  CompareCost( $v, current$ )
9    if  $g(current) + Cost(current, v) < g(v)$  then    // path1
10      $g(v) \leftarrow g(current) + Cost(current, v);$ 
11      $Parent(v) \leftarrow current;$ 
12     [ $UpdateParent(v);$ ]    // Theta*
13  UpdateParent( $v$ )
14     $grandpa \leftarrow Parent(Parent(v));$ 
15    if  $LineofSight(v, grandpa)$  then
16     if  $g(grandpa) + Cost(grandpa, v) < g(v)$  then    //path2
17        $g(v) \leftarrow g(grandpa) + Cost(grandpa, v);$ 
18        $Parent(v) \leftarrow grandpa;$ 
19  return  $v$ 

```

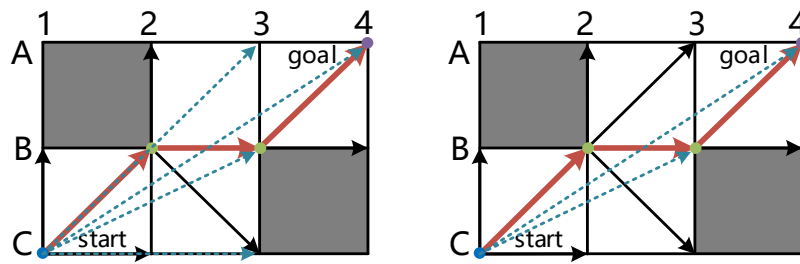


Figure 2. The extension processes of Theta* (left) and Lazy Theta* (right): the dashed lines are line-of-sight paths. Lazy Theta* performs less LoS-Check than Theta*.

3.2. Lazy Theta*

Different from Theta*, Lazy Theta* only performs LoS-Check on the current vertex that is waiting to be extended, but not every newly-generated vertex. Because many generated vertices will have no chance of being extended, there is no use to perform LoS-Check on them. Lazy Theta* delays the LoS-Check until the vertex is to be extended, which will reduce the check amount, speeding up the planning process, while continuing to find the close-to-shortest path. As Figure 2 (right) shows, there are only two times of LoS-Checks in the extension process of Lazy Theta*, and they are performed on the vertices B3 and A4 with their grandparent vertices. Activate the code in Line 5 of Algorithm 1, and the algorithm will become Lazy Theta*.

4. LLA* with Prioritized Trees

The lazy evaluation strategy in Lazy Theta* reduces LoS-Check greatly, saving much computation. Actually, the delayed strategy was firstly used in Lazy PRM very early [43], which delays the collision detection until it is truly necessary. Lazy Weighted A* [55] also takes the lazy evaluation, which focuses on reducing the number of times checking the expensive-to-evaluate edges. We are inspired by Lazy Theta* and present Late LoS-Check A* (LLA*) to further delay the LoS-Check operation. However, if we only reduce the amount of LoS-Check, the generated path will be longer. Therefore, we introduce the prioritized trees to update discriminatively the g values of different successors after LoS-Check.

The main idea of LLA* (Algorithm 3) has two parts: reducing LoS-Check more and updating the g value with discrimination. Assume the current vertex is v and the parent vertex of v is u . Furthermore, we define the grandparent vertex of u as gp . To reduce LoS-Check more, we delay the LoS-Check between u and its grandparent vertex gp until v is chosen to be extended. When v is to be extended, we firstly perform LoS-Check between u and gp , and if the line-of-sight exists, set gp as the parent of u . Updating the g value with discrimination means that we only update the g value of the current vertex v with the new $g(u)$, but not the g value of every successor of u . As a result, among all the successors of u , vertex v will own even smaller cost, meaning that the successors of v will also have the priority to be extended.

As shown in Figure 3, when vertex v_5 is to be extended, LLA* firstly updates the parent of v_3 . There exists the line-of-sight between v_3 and its grandparent vertex v_{start} , so v_{start} becomes the parent of v_3 , and the g value of v_3 is updated. Then, LLA* will just update the g value of v_5 (the only successor of v_3 chosen to be extended) and ignore other successors (without updating their g values). That is to say, the real cost from v_{start} to v_5 , which is also denoted as $g(v_5)$, will be even smaller than $g(v_4)$ and $g(v_6)$, being as g values of other successors of v_3 . As a result, the successors of v_5 may be chosen with priority to be extended, which can speed up the planning toward the goal.

Algorithm 3: LLA*

```

1   $g(v_{start}) \leftarrow 0;$      $closed \leftarrow \emptyset;$      $open \leftarrow \emptyset;$ 

2   $open.Add(v_{start}, g(v_{start}) + h(v_{start}));$ 

3  while  $open \neq \emptyset$  do

4     $current \leftarrow open.Pop\_least();$     // pop the vertex with the least  $f$  value

5     $pa \leftarrow Parent(current)$ 

6     $pa \leftarrow UpdateParent(pa);$     // late LoS-Check

7     $g(current) \leftarrow g(pa) + Cost(pa, current)$     // discriminatory updating

8    if  $current == goal$  then

9      return  $path;$ 

10    $closed.Add(current);$ 

11   for each  $v \in Succ(current)$  do

12     if  $v \notin closed$  then

13       if  $v \notin open$  then

14          $g(v) \leftarrow \infty;$ 

15          $Parent(v) \leftarrow NULL;$ 

16          $UpdateNode(v, current);$ 

```

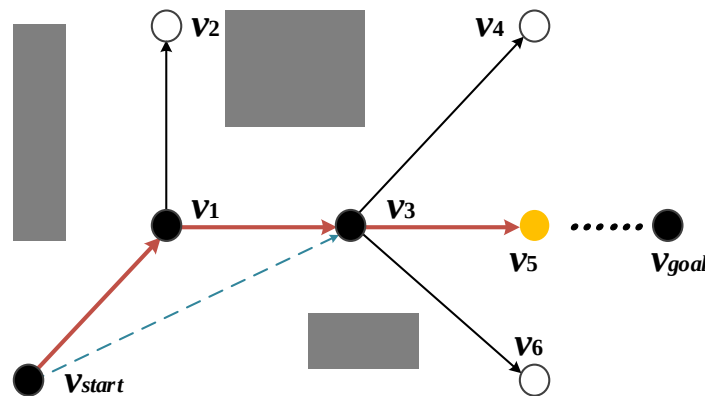


Figure 3. The planning process of LLA* (v_5 is to be extended). Firstly, perform LoS-Check between v_3 (parent of v_5) and v_{start} (grandparent of v_3). Then, use the updated $g(v_3)$ to only update $g(v_5)$, making $g(v_5)$ even less than $g(v_4)$ and $g(v_6)$.

In facts, Theta* and Lazy Theta* treat different successors of the same vertex equally. For example, in Theta* or Lazy Theta*, the set of $\{v_4, v_5, v_6\}$ is all the successors of v_3 , and they get the same g value from v_3 . However, in LLA*, we want to propagate a smaller g value from v_3 to v_5 because v_5 has a smaller h value, which can reduce the total cost more $f(v_5) = g(v_5) + h(v_5) = g(v_3) + Cost(v_3, v_5) + h(v_5)$. It can be foreseen that the successors of v_5 may be extended with priority compared to the successors of v_4 and v_6 , which will speed up the planning.

In Section 3, we mentioned that A* algorithm is both resolution complete and resolution optimal, which means that it could find the optimal path on the grids. However, A* cannot find the real shortest path because the path is constrained to the grid edges. Meanwhile, LoS-Check connects two non-adjacent vertices, which could relax the constraints to grid edges and finally generate shorter paths. Theta* employs LoS-Check, and it could find shorter paths than A*, but it is not optimal.

If we want to find the optimal solution in a graph, it is necessary to guarantee that the solution cost estimate (i.e., the f value) from a parent to its child state must be monotonically increasing [34]. Thus, the expansion of the new state should be based on the monotone condition, which ensures that the final path is optimal upon the given graph. The g value of states in the Theta* algorithm may be decreased, which does not satisfy the monotone condition. Thus, Theta* is not optimal. The Lazy Theta* and LLA* algorithms are both variants of Theta*, which are not optimal either. However, “non-optimal” does not mean that the algorithm cannot generate shorter paths.

The key idea of LLA* is further delaying the LoS-Check operation of a vertex until its successor expands and discriminatively updates the g values. LLA* is complete and could always find better paths than A*. For example, in each iteration, with the help of LoS-Check, LLA* will result in a smaller cost for the current state (the state that is chosen to expand new states) when compared with A*. Then, the original state with the least f value in the *open* list of A* will now get an even smaller cost in LLA*. LLA* does not change the framework of A*, and the worst case for LLA* is degrading into A*.

In fact, LoS-Check is a greedy strategy; it connects two states if the path cost could become smaller. However, in some extreme conditions, such as the corner of the blocked cells, LoS-Check will not work because the line-of-sight is blocked by the obstacles [2]. LLA* reduces the amount of LoS-Check, which gains a trade-off between the execution time and path length. LoS-Check is a kind of long-distance collision detection, which will cost a longer time than the simple check. LLA* employs less LoS-Check than Theta* and Lazy Theta*, so it runs faster. Because of less LoS-Check, LLA* employs a weaker greedy strategy to some degree, which will not be trapped in the extreme conditions at the obstacle corner. Thus, LLA* could find better paths than Lazy Theta* on average and even find better paths than Theta* sometimes (we will show an example in Figure 6).

Moreover, the discriminatory updating used in the LLA* algorithm is also a kind of greedy strategy, and it truly works. After LoS-Check, LLA* only updates the g value of the successor state, which has the potential to generate better solutions. This operation could accelerate the planning procedure, and the worst case is degrading into searching paths as A* does. In other words, if the greedy search methods fail, LLA* will still search the paths like A*. Therefore, the LLA* algorithm will run faster than A* on average.

5. LoS-Slider Smoothing

The path of A* is often constrained to the grid edges, which looks unrealistic. LoS-Check has been used for post-processing (smoothing) in path planning [56], and it can make the path of A* much shorter. In this paper, we design a smoothing algorithm called Line-of-Sight Slider (LoSS), and this kind of post-processing method will not cost much time.

According to the moving direction of the current node, the LoSS algorithm is divided into two types: backward-LoSS and forward-LoSS. In a given path, every node has only one parent node and only one successor node. When the current node moves from the goal node to the start node, it is called the backward-LoSS algorithm, and when the current node moves from the start node to the goal node, it is called the forward-LoSS algorithm. When the *current* node moving direction is determined, its successor node, which is called *child*, and the child's successor, which is called *grandchild*, will be both determined as well. At this time, the *slider* node recurrently slides from node *grandchild* to node *child* by a small stepsize, and LoS-Check is performed between the *slider* node and the *current* node each time. When it is successful, stop the *slider* from sliding, and set the *slider* node as the successor node of the *current* node. Subsequently, set the successor of the *current* node as the new *current* node for the next round of checking.

Figure 4 (top) shows the backward-LoSS diagram; the node C is selected as the current node, which moves from the goal node to the start node, with the node B as the child of C and the node A as the grandchild of C. According to the rule of LoSS, the slider node slides from A to B, and after one iteration, the LoS condition between C and slider is satisfied, meaning that this round of smoothing is complete. As a result, the node slider becomes the successor node of C. It can be found that the length of the newly-generated route {A, slider, C} is shorter than that of the original path {A, B, C}. For the forward-LoSS of Figure 4 (bottom), the principle is the same as backward-LoSS, with only the opposite direction. The specific implementation of the LoSS algorithm can be seen in Algorithm 4, which can be used several times to get even shorter paths. In Line 10 of Algorithm 4, the function Slide_From_To(slider, child, stepsize) is based on the linear interpolation of the segment between the slider and child. The parameter stepsize decides the interpolation resolution. The interpolation generates many intermediate points, and the slider will move to these points one by one. The moving direction is set from the grandchild to the child, and the slider will be updated each time. If we set the parameter *stepsize* as a smaller value, the final smoothed path will have a shorter length.

Algorithm 4: LoSS

Input: $path = \{s_1, s_2, s_3, \dots, s_n\}$

```

1  current  $\leftarrow s_1$ ;  new_path  $\leftarrow \{current\}$ ;
2  child  $\leftarrow Succ(current)$ ;    // the successor in path
3  grandchild  $\leftarrow Succ(child)$ ;
4  while grandchild  $\neq NULL$  do
5    slider  $\leftarrow grandchild$ ;
6    while slider  $\neq child$  do
7      if LineoSight (slider, current) then
8        Succ(slider)  $\leftarrow grandchild$ ;
9        Succ(current)  $\leftarrow slider$ ;
10     break;
11     slider  $\leftarrow Slide\_From\_To(slider, child, stepsize)$ ;
12   current  $\leftarrow Succ(current)$ ;
13   new_path  $\leftarrow new\_path \cup \{current\}$ ;
14   child  $\leftarrow Succ(current)$ ;
15   grandchild  $\leftarrow Succ(child)$ ;
16 return new_path;
  
```

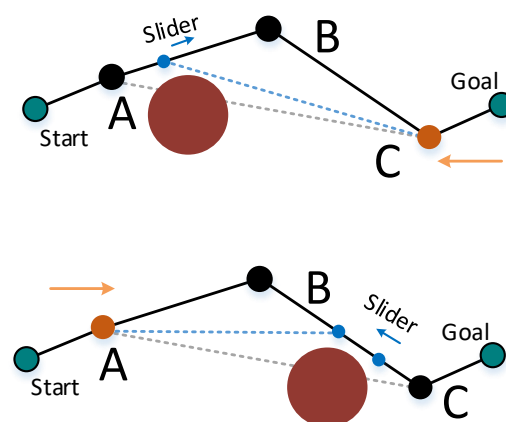


Figure 4. Backward-LoSS (top) and forward-LoSS (bottom). The slider node recurrently slides from the grandchild of the current node to the child of the current node by a small stepsize. In backward-LoSS, the current node moves from the goal node to the start node. In forward-LoSS, the current node moves from the start node to the goal node.

6. Experiment

We compared A*, Theta*, Lazy Theta*, and LLA* on some pathfinding benchmark maps [2,57], which contain game maps, maze maps, random maps, and street maps (Figure 5). Each type of map has different scenarios, and each scenario has different pairs of start and goal points. We also implemented LLA* and the other three planners on the basis of the Open Motion Planning Library (OMPL) [51] and compared these planners in some virtual scenes.

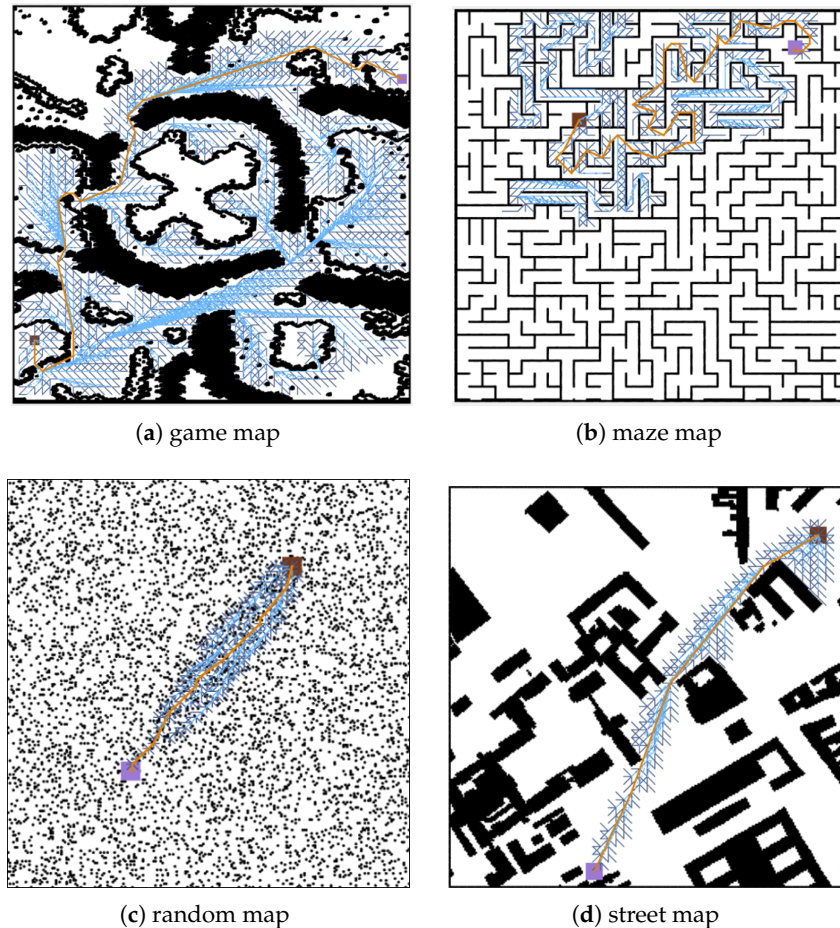


Figure 5. Extension processes of LLA* on four different kinds of benchmark maps: game map (a), maze map (b), random map (c), and street map (d). Orange lines represent the final paths, and the blue lines are new edges after LoS-Check.

6.1. Benchmark Test

The pathfinding benchmark maps have different sizes, and the units of the grid maps are pixels. The size of game maps varies from 22×28 – 1260×1104 . Maze maps (with a size of 512×512) contain different corridor widths of mazes, and random maps (with a size of 512×512) are covered by randomly-blocked cells with varying percentages. Street maps are the thumbnails of the real streets in different cities, and their sizes range from 256×256 – 1024×1024 . We implemented these algorithms on the basis of Weighted A* with $f(n) = (1 - \lambda) * g(n) + \lambda * h(n)$, $\lambda \in [0, 1]$. In the experiment, there were over ten thousand tests done on each kind of map, and we set three different values for λ . Figure 6 shows the extension processes of A*, Theta*, Lazy Theta*, and LLA* on a street map. In this example, LLA* was the fastest among the four algorithms, and it also generated the shortest path. Of course, this is just a particular case, and a general evaluation is given in the benchmark tests.

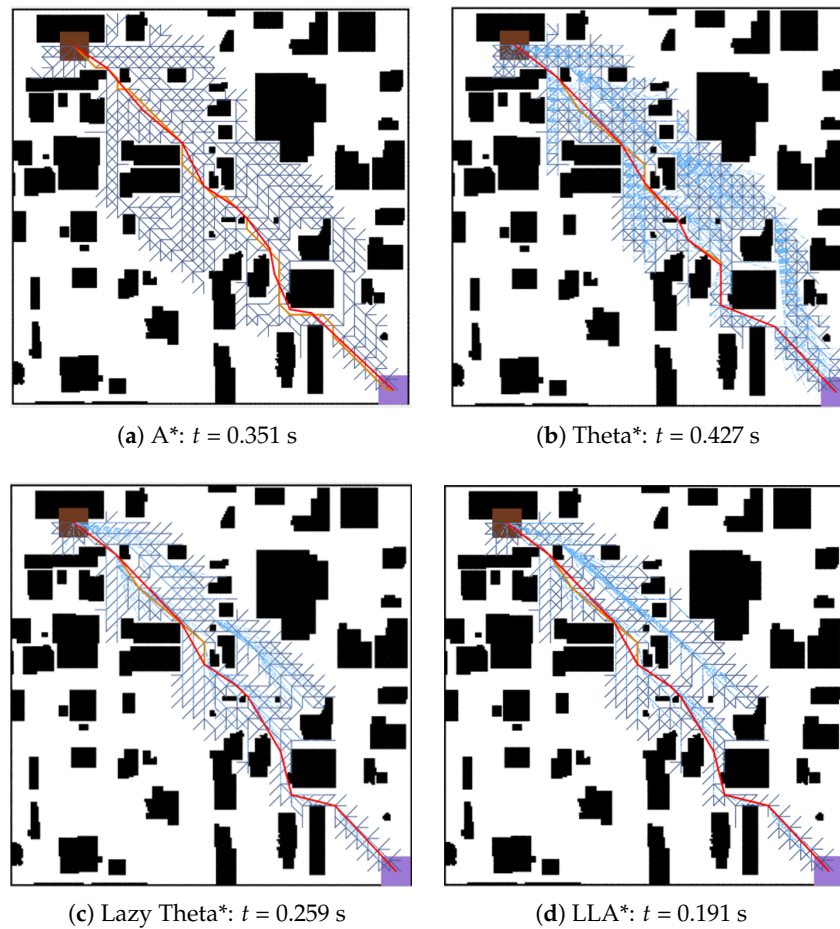


Figure 6. Extension processes of A* (a), Theta* (b), Lazy Theta* (c), and LLA* (d) on a street map. Orange lines are initial paths, and red lines are smoothed paths after the LoSS smoothing (one time of backward-LoSS and one time of forward-LoSS). (a) A*: initial length is 336.68, smoothed length is 320.35; (b) Theta*: initial length is 322.51, smoothed length is 317.79; (c) Lazy Theta*: initial length is 320.95, smoothed length is 316.31; (d) LLA*: initial length is 319.92, smoothed length is 316.31. In this example, LLA* has the least execution time and path length, which is a particular case, and the general evaluation is shown in the benchmark tests.

Figure 7 shows the planning results against different values of λ in the benchmark tests. The average path length and execution time of the four algorithms on every map is shown in Table 1. When the factor λ equaled 0.5, these algorithms were all based on the original A*, and we can see that the path length of LLA* was shorter than that of Lazy Theta* or A*. Moreover, the execution time of LLA* was the least among the four algorithms. Meanwhile, we can also find that the paths of LoS-Check-based planners (Theta*, Lazy Theta*, and LLA*) were not shortened very much after LoSS smoothing, but the path of A* was shortened obviously. Therefore, post-smoothing with LoS-Check was also effective in generating shorter paths for the planners that did not use LoS-Check. However, the effects of post-processing were not as good as those of online LoS-Check.

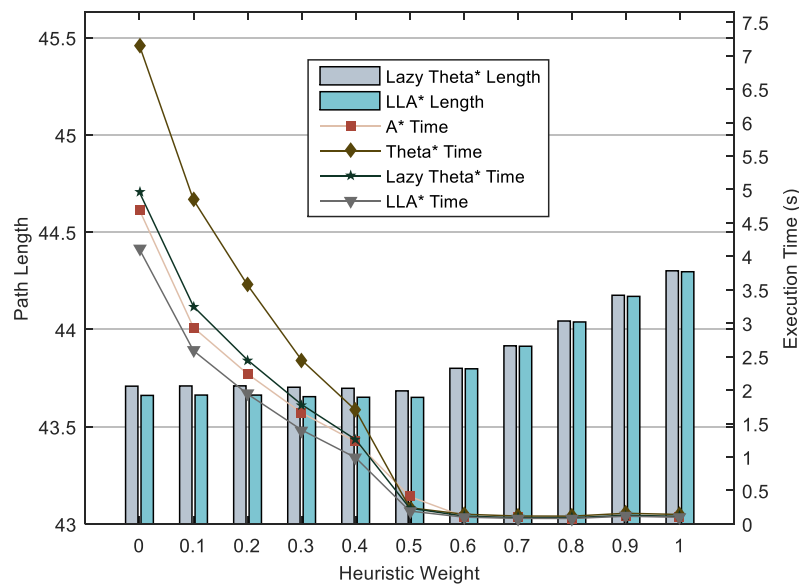


Figure 7. The average path length and execution time plotted against the heuristic weight. LLA* is the fastest, and it generates a shorter path than Lazy Theta*.

From both Figure 7 and Table 1, we can see that when the factor $\lambda > 0.5$, the path length of Lazy Theta* and LLA* was similar, which was slightly longer than that of Theta*, but LLA* was still the fastest. Meanwhile, the gap of the execution time between LLA* and the other algorithms was not as big as the gap when $\lambda = 0.5$. Moreover, when the factor $\lambda < 0.5$, the path length of LLA* kept being shorter than that of Lazy Theta* or A*, and LLA* kept being the fastest among the four algorithms. At the same time, the gap of the execution time between LLA* and Theta* was bigger than the gap when $\lambda = 0.5$. We can see that LLA* always kept being the fastest and usually resulted in a shorter path than Lazy Theta* (just slightly longer than Theta*) no matter what the factor λ would be.

Table 1. Average path length and execution time (s) with/without LoSS smoothing.

λ	Algorithms	LoSS	Maps							
			Game		Maze		Random		Street	
			Length	Time	Length	Time	Length	Time	Length	Time
0.3	A*	No	45.63	1.853	45.98	0.803	42.21	1.588	49.59	2.009
		Yes	43.33	1.869	44.14	0.823	40.99	1.609	47.14	2.027
	Theta*	No	43.31	2.804	44.08	1.207	40.35	2.022	47.11	3.051
		Yes	43.31	2.815	44.06	1.221	40.32	2.038	47.11	3.063
	Lazy Theta*	No	43.35	1.993	44.20	0.884	40.73	1.671	47.19	2.171
		Yes	43.34	2.004	44.16	0.897	40.66	1.687	47.16	2.182
	LLA*	No	43.33	1.561	44.15	0.686	40.62	1.357	47.14	1.699
		Yes	43.32	1.572	44.12	0.699	40.56	1.372	47.13	1.712

Table 1. Cont.

λ	Algorithms	LoSS	Maps							
			Game		Maze		Random		Street	
			Length	Time	Length	Time	Length	Time	Length	Time
0.5	A*	No	45.64	0.437	45.98	0.409	42.21	0.321	49.59	0.416
		Yes	43.35	0.455	44.14	0.432	41.03	0.344	47.15	0.434
	Theta*	No	43.32	0.191	44.09	0.431	40.42	0.232	47.11	0.215
		Yes	43.32	0.202	44.06	0.446	40.36	0.248	47.11	0.226
	Lazy Theta*	No	43.35	0.201	44.18	0.366	40.65	0.227	47.18	0.216
		Yes	43.34	0.211	44.12	0.381	40.55	0.243	47.15	0.227
	LLA*	No	43.33	0.168	44.15	0.291	40.58	0.181	47.14	0.176
		Yes	43.32	0.178	44.09	0.306	40.49	0.197	47.12	0.187
	A*	No	45.76	0.074	46.73	0.161	42.92	0.069	49.93	0.084
		Yes	43.36	0.085	44.23	0.176	41.22	0.086	47.18	0.097
0.7	Theta*	No	43.38	0.101	44.31	0.214	41.60	0.078	47.25	0.112
		Yes	43.36	0.106	44.18	0.224	41.37	0.091	47.19	0.121
	Lazy Theta*	No	43.39	0.078	44.34	0.169	41.68	0.071	47.27	0.087
		Yes	43.36	0.084	44.21	0.179	41.42	0.082	47.21	0.095
	LLA*	No	43.39	0.071	44.33	0.141	41.68	0.064	47.27	0.078
		Yes	43.36	0.076	44.20	0.151	41.43	0.076	47.21	0.086

6.2. Complex Street Test

In the previous subsection, we did some benchmark tests on various maps. Each type of map had different scenarios, and each scenario had different pairs of start and goal points. The average planning results, including path length and execution time, showed that the LLA* algorithm ran the fastest among the four given planners, and it generated shorter paths than Lazy Theta* and A*.

It seemed that LLA* had a good performance in both the planning speed and quality. However, the average results could not display the performance of LLA* in some special situations. Thus, we selected six complex street maps from the benchmark maps mentioned in Section 6.1, which are shown in Figure 8. These maps included thumbnails of the streets in Berlin, Boston, London, Milan, Moscow, and Paris, which were all of a large size, 1024×1024 . These complex street tests contained difficult tasks, where the start position and goal position were both set at some narrow corners. We performed this kind of test to explore the extreme cases and limits of these planners.

Table 2 shows the planning results of the four planners on the six complex street maps, including path length, execution time, and LoS-Check number. In the original design, we aimed to speed up the planning algorithm by reducing the LoS-Check number. Therefore, we provided the LoS-Check number to reflect the relationship between execution time and the line-of-sight check operation. From the planning results on the six complex city street maps, we can find that LLA* still ran the fastest among the four planners, and it always generated shorter paths than Lazy Theta* and A*. Moreover, it is interesting that LLA* sometimes generated shorter paths than Theta*, such as the planning results of the street tests in Milan and Paris (this is similar to the results in Figure 6). LLA* performed the least LoS-Check, so it ran faster than the other planners. Meanwhile, because of using the discriminatory updating strategy in the planning process, it also generated shorter paths than Lazy Theta* and sometimes generated shorter paths than Theta*. The discriminatory updating strategy helped the planner focus on exploring the states that have the potential to improve the final paths.

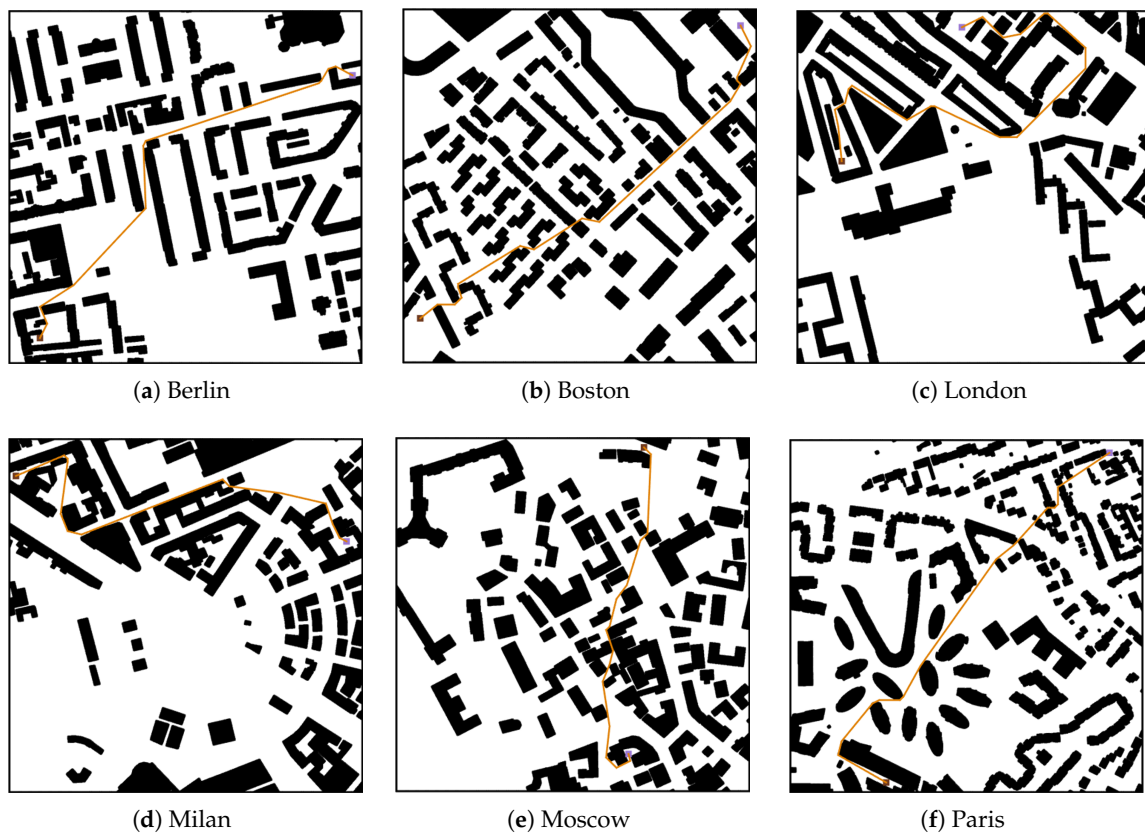


Figure 8. The path planning results using the LLA* algorithm on six complex street maps. These street maps were chosen from the benchmark maps in Section 6.1, and they are thumbnails of the real streets in six different cities. The size of these maps is 1024×1024 . The start position and goal position are both set at some narrow corners, making the planning task more difficult to solve. The final paths are shown as orange solid lines.

Table 2. Path planning results on six complex street maps.

Complex Street Test		City					
		Berlin	Boston	London	Milan	Moscow	Paris
A*	Length	1456.95	1465.51	1640.66	1395.68	1110.97	1462.37
	Time (s)	0.116	0.137	0.142	0.075	0.106	0.062
	LoS-Check	0	0	0	0	0	0
Theta*	Length	1377.12	1416.24	1563.75	1308.36	1056.04	1402.11
	Time (s)	0.269	0.276	0.301	0.138	0.183	0.097
	LoS-Check	20240	21766	27027	12561	17841	8714
Lazy Theta*	Length	1389.07	1428.86	1593.02	1312.56	1063.10	1407.96
	Time (s)	0.134	0.153	0.179	0.076	0.102	0.061
	LoS-Check	3044	3156	3752	1679	2476	1369
LLA*	Length	1381.41	1418.19	1573.08	1306.33	1057.57	1401.32
	Time (s)	0.112	0.106	0.131	0.062	0.076	0.045
	LoS-Check	2347	2136	2628	1314	1737	958

The six complex street tests showed that the LLA* algorithm still outperformed Lazy Theta* and A*, even in some extreme cases. Until now, both the general benchmark test and the complex street test showed that LLA* had a good and robust performance in various environments. We will continue to verify LLA* in some virtual scenes in the next subsection.

6.3. Virtual Scenes

We implemented the four planners on the basis of the famous motion planning library, OMPL [51]. It provides low-level data structures and modules, such as the state generation and collision detection, which are essential for high-level motion planning algorithms. Moreover, it also collects a known set of benchmark problems for the comparison of different planners. In practice, OMPL could cooperate with ROS (Robot Operating System) [58], which plays an important role in robotic research and industry. Figure 1 shows a classic scene of the piano mover problem. Figure 9 and Figure 11 show the planning trajectories of the four algorithms on the maze map #1 and #2, respectively, and the size of the two maps is 90×90 .



Figure 9. The planning results for an animated character using A* (a), Theta* (b), Lazy Theta* (c), and LLA* (d) on Maze #1. The extension details and final composite moving trajectories are shown in red color.

Table 3 shows several parameters of the planning results of the maze test #1 in Figure 9, including the number of generated states, LoS-Check amount, simple-check amount, path length, and execution time. We did 500-times testing on the maze scenario, and the execution time for each test is shown in Figure 10. The simple-check represents the collision check between two neighbor vertices that are close to each other. Considering that LoS-Check may be performed between two vertices that are far

away from each other, the time spent on the simple-check was shorter than that on LoS-Check on average. Therefore, more LoS-Check will cost more execution time, which is what we want to reduce. It is easy to find that LLA* cost the least time to complete the path planning, which was followed by Lazy Theta*, A*, and Theta*. Meanwhile, the path cost of LLA* was shorter than that of A* and Lazy Theta*, but slightly longer than that of Theta*.

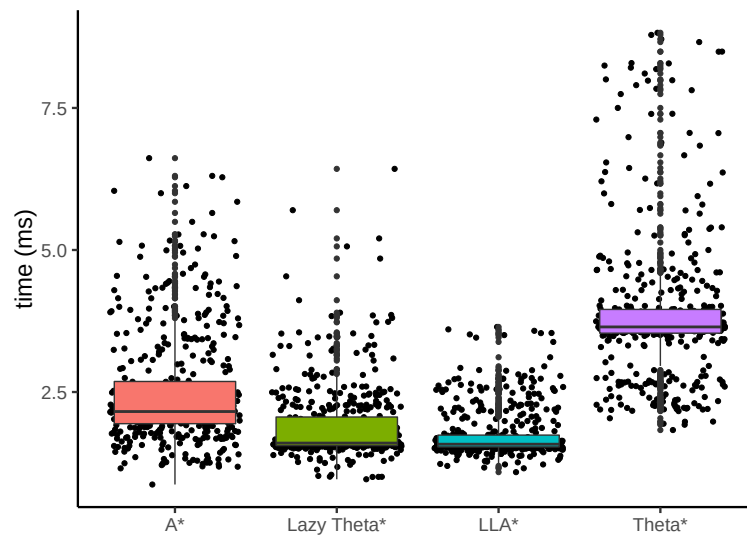


Figure 10. The execution time of 500 times of the maze test #1 using the four planners. LLA* is the fastest among these planners on average.

Table 3 shows that Theta* performed LoS-Check 6027 times, while LLA* performed it 2451 times. It is easy to understand that the path cost of LLA* was longer than that of Theta*, but the execution time was the opposite because Theta* performed more LoS-Check operations. Meanwhile, when comparing with Lazy Theta*, LLA* generated fewer states and performed less LoS-Check and simple-check. Therefore, LLA* was faster than Lazy Theta*. However, it is interesting that the path cost of LLA* was also less than that of Lazy Theta*. Moreover, because LLA* performed less simple collision detections than A*, LLA* was slightly faster than A* as well. In brief, LLA* behaved well in both the path length and the planning speed.

Table 3. Planning results of the maze test #1.

	A*	Theta*	Lazy Theta*	LLA*
states	2831	2681	2681	2676
LoS-Check	0	6027	2609	2451
simple-check	8186	3133	7699	7290
path cost	127.38	125.26	125.89	125.52
time (ms)	2.335	4.191	2.240	1.779

Table 4 shows the results of the maze test #2 in Figure 11. We also did 500-times testing on this maze scenario, and the execution time for each test is shown in Figure 12. The planning results in the maze test #2 were in accord with those in the maze test #1. LLA* was still the fastest among the four planners.

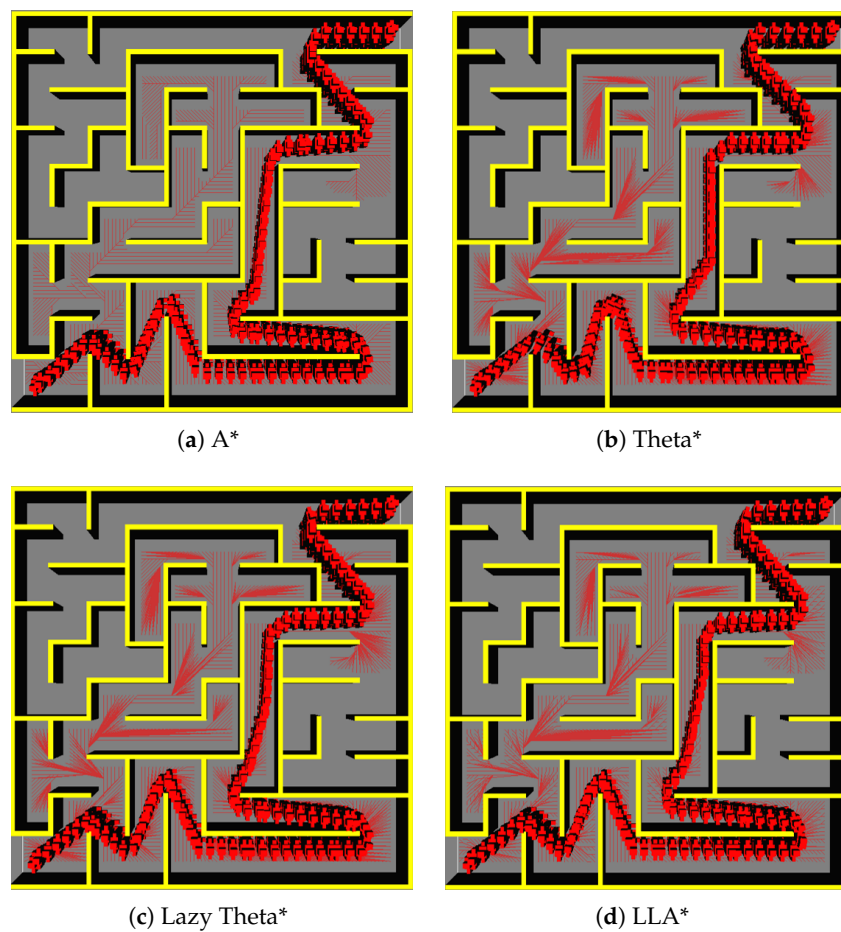


Figure 11. The planning results for an animated character using A* (a), Theta* (b), Lazy Theta* (c), and LLA* (d) on Maze #2. The extension details and final composite moving trajectories are shown in red color.

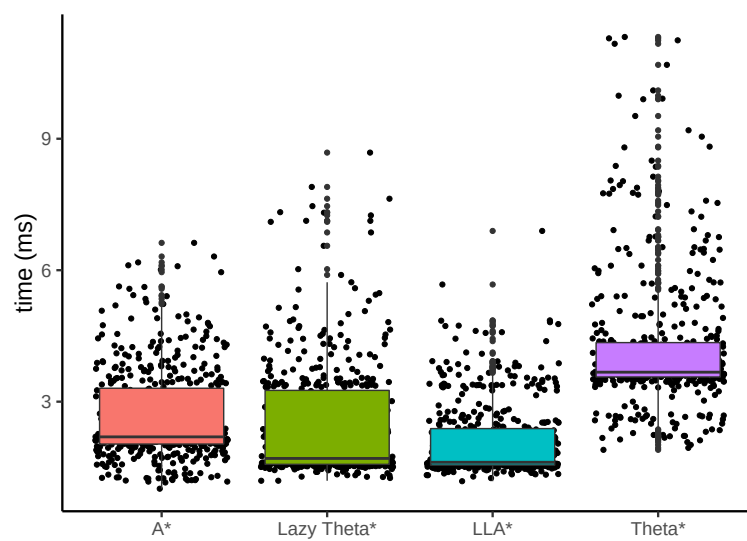


Figure 12. The execution time of 500 times of the maze test #2 using the four planners. LLA* is the fastest among these planners on average.

Table 4. Planning results of the maze test #2.

	A*	Theta*	Lazy Theta*	LLA*
states	3654	3616	3615	3592
LoS-Check	0	8114	3545	3478
simple-check	10232	4080	10161	9747
path cost	281.75	274.31	274.63	274.58
time (ms)	2.639	4.218	2.432	2.091

6.4. Results and Analysis

From the experiments in the previous three subsections, including the benchmark test, the complex street test, and the test in virtual scenes, we can find that LLA* ran the fastest among the four given planners, and it generated shorter paths than the Lazy Theta* and A* algorithms. LLA* employed the discriminatory updating strategy to explore the states that had the potential to improve the final solutions with priority, and this strategy was helpful in accelerating the planning algorithms.

The planning results in the three tests showed that LLA* performed the least LoS-Check, and this could explain why LLA* ran the fastest, especially on the large size of the maps with a complex environment. Both the lazy evaluation strategy and the discriminatory updating strategy led to less LoS-Check, which sped up the planning procedure. Actually, LLA* was resolution complete like A*, which was bound to find a solution if it really existed.

LoS-Check connected a state with its grandparent, but it did not work at a corner of the blocked cell because the line-of-sight was blocked by the obstacles. Meanwhile, LLA* further delayed LoS-Check, which reduced the risk that the line-of-sight was blocked. Thus, LLA* generated shorter paths than Lazy Theta* and sometimes generated shorter paths than Theta*. Of course, with the help of LoS-Check, LLA* always found a better solution than A*. LLA* did not change the framework of A*, and the worst case for LLA* was degrading into A*.

Overall, the LLA* algorithm showed a good and robust performance in path planning, which outperformed Theta*, Lazy Theta*, and A* in the aspect of planning speed. LLA* also generated shorter paths than Lazy Theta* and A*. However, the planning algorithms that used LoS-Check, such as LLA*, Theta*, and Lazy Theta*, were not suitable in nonholonomic planning. In nonholonomic planning, if LoS-Check were used, some states in the configuration space would not be explored because of constraints (e.g., angle constraint), making the planning algorithm not complete. Thus, it is an interesting research topic how to employ path shortcutting methods in nonholonomic planning.

7. Conclusions

Line-of-Sight Check (LoS-Check) is efficient for any-angle path planning, which has been successfully used in Theta*. In the high-dimensional environment, there will be too many LoS-Checks in Theta*, which may cost much computation. Lazy Theta* introduces delayed evaluation into Theta*, reducing the LoS-Check amount obviously. Inspired by Lazy Theta*, we presented Late LoS-Check A* (LLA*) to further delay the LoS-Check and update the g value of different successor vertices with discrimination, which is different from Lazy Theta* or Theta*. If we just delayed LoS-Check, the path planned by LLA* would never be shorter than that of Lazy Theta*. Therefore, the key of LLA* was introducing the discriminatory strategy in LoS-Check operations. It could further reduce the real cost (g value) of the node that is chosen to be extended.

We implemented LLA* on the basis of a tree structure, and it was especially useful for planning under differential constraints. LoS-Check is performed between two non-adjacent vertices in grid-based path planning, which will cost much time with the map size increasing. LLA* updates the cost of the vertex, which has a better estimation of the total path cost, and this is based on the structure of prioritized trees. LLA* is not strictly optimal, but truly results in a shorter path than A* and Lazy Theta*. The experiment showed that LLA* ran faster and planned a shorter path than Lazy

Theta*; furthermore, the path planned by LLA* was just slightly longer than that of Theta*, while costing less time. The algorithms in this paper was integrated in the Open Motion Planning Library (OMPL), which could be used in practice.

Our future work will be focused on optimal planning algorithms and multi-agent path planning. In addition, we will improve the LLA* algorithm and apply it to nonholonomic path planning as well. This paper gives an attempt to employ LoS-Check with discriminatory updating in the grid-based A* algorithm, and it could be applied in other variants of A* as well. We hope this kind of discriminatory principle could give some reference and inspiration to peers in the field of robotic planning.

Author Contributions: Conceptualization, C.Z. and Y.T.; methodology, C.Z.; writing—original draft preparation, C.Z.; writing—review and editing, Y.T.; supervision, H.L.; project administration, L.Z.

Funding: This research was funded by the National Natural Science Foundation of China under Grant 61602496.

Acknowledgments: We would like to thank the developers of the Open Motion Planning Library (OMPL) for their open source codes and benchmark scenarios.

Conflicts of Interest: The authors declare no conflict of interest.

Abbreviations

The following abbreviations are used in this manuscript:

LoS-Check	Line-of-Sight Check
LLA*	Late LoS-Check A*
RRT	Rapidly-exploring Random Tree
PRM	Probabilistic Road Map
APF	Artificial Potential Field
RL	Reinforcement Learning
MDP	Markov Decision Process
DRL	Deep Reinforcement Learning
DQN	Deep Q-Network
VIN	Value Iteration Network
VPN	Value Prediction Network
PST	Precomputed Search Trees
RTS	Real-Time Strategy
FMT*	Fast Marching Trees
LBT-RRT	Lower Bound Tree-RRT
BIT*	Batch Informed Trees
LPA*	Lifelong Planning A*
OMPL	Open Motion Planning Library
WA*	Weighted A*
LoSS	Line-of-Sight Slider
ROS	Robot Operating System

References

1. Choset, H.M.; Hutchinson, S.; Lynch, K.M.; Kantor, G.; Burgard, W.; Kavraki, L.E.; Thrun, S. *Principles of Robot Motion: Theory, Algorithms, and Implementation*; MIT Press: Cambridge, MA, USA, 2005.
2. Nash, A.; Daniel, K.; Koenig, S.; Felner, A. Theta*: Any-angle path planning on grids. In *Proceeding of the AAAI'07 Proceedings of the 22nd National Conference on Artificial Intelligence*, Vancouver, BC, Canada, 22–26 July 2007; Volume 7, pp. 1177–1183.
3. Qin, L.; Hu, Y.; Yin, Q.; Zeng, J. Speed Optimization for Incremental Updating of Grid-based Distance Maps. *Appl. Sci.* **2019**, *9*, 2029. [[CrossRef](#)]

4. Cui, M.; Harabor, D.D.; Grastien, A. Compromise-free Pathfinding on a Navigation Mesh. In Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence (IJCAI-17), Melbourne, Australia, 19–25 August 2017; pp. 496–502.
5. Bellman, R. The theory of dynamic programming. *Bull. Am. Math. Soc.* **1954**, *60*, 503–515. [[CrossRef](#)]
6. Dijkstra, E.W. A note on two problems in connexion with graphs. *Numer. Math.* **1959**, *1*, 269–271. [[CrossRef](#)]
7. Hart, P.E.; Nilsson, N.J.; Raphael, B. A formal basis for the heuristic determination of minimum cost paths. *IEEE Trans. Syst. Sci. Cybern.* **1968**, *4*, 100–107. [[CrossRef](#)]
8. Zhang, C.; Zhou, L.; Liu, H. LaLo-Check: A Path Optimization Framework for Sampling-Based Motion Planning with Tree Structure. *IEEE Access* **2019**, *7*, 100733–100746. [[CrossRef](#)]
9. LaValle, S.M.; Kuffner, J.J., Jr. Randomized kinodynamic planning. *Int. J. Robot. Res.* **2001**, *20*, 378–400. [[CrossRef](#)]
10. Kavraki, L.E.; Svestka, P.; Latombe, J.C.; Overmars, M.H. Probabilistic Roadmaps for Path Planning in High-Dimensional Configuration Spaces. *IEEE Trans. Robot. Autom.* **1996**, *12*, 566–580. [[CrossRef](#)]
11. Karaman, S.; Frazzoli, E. Sampling-based algorithms for optimal motion planning. *Int. J. Robot. Res.* **2011**, *30*, 846–894. [[CrossRef](#)]
12. Wei, K.; Ren, B. A method on dynamic path planning for robotic manipulator autonomous obstacle avoidance based on an improved RRT algorithm. *Sensors* **2018**, *18*, 571. [[CrossRef](#)]
13. Gawron, T.; Michałek, M. A G3-Continuous Extend Procedure for Path Planning of Mobile Robots with Limited Motion Curvature and State Constraints. *Appl. Sci.* **2018**, *8*, 2127. [[CrossRef](#)]
14. Khatib, O. Real-time obstacle avoidance for manipulators and mobile robots. In *Autonomous Robot Vehicles*; Springer: Cham, Switzerland, 1986; pp. 396–404.
15. Sutton, R.S.; Barto, A.G. *Reinforcement Learning: An Introduction*; MIT Press: Cambridge, MA, USA, 2018.
16. Bellman, R. Dynamic programming. *Science* **1966**, *153*, 34–37. [[CrossRef](#)]
17. Puterman, M.L. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*; John Wiley & Sons: Hoboken, NJ, USA, 2014.
18. Bertsekas, D. *Dynamic Programming and Optimal Control*; Athena Scientific: Belmont, MA, USA, 2012; Volume II.
19. Watkins, C.J.; Dayan, P. Q-learning. *Mach. Learn.* **1992**, *8*, 279–292. [[CrossRef](#)]
20. Mnih, V.; Kavukcuoglu, K.; Silver, D.; Rusu, A.A.; Veness, J.; Bellemare, M.G.; Graves, A.; Riedmiller, M.; Fidjeland, A.K.; Ostrovski, G.; et al. Human-level control through deep reinforcement learning. *Nature* **2015**, *518*, 529. [[CrossRef](#)]
21. Tamar, A.; Wu, Y.; Thomas, G.; Levine, S.; Abbeel, P. Value iteration networks. In Proceedings of the Advances in Neural Information Processing Systems Conference, Barcelona, Spain, 5–11 December 2016; pp. 2154–2162.
22. Oh, J.; Singh, S.; Lee, H. Value prediction network. In Proceedings of the Advances in Neural Information Processing Systems Conference, Long Beach, CA, USA, 4–9 December 2017; pp. 6118–6128.
23. Kuffner, J.; Latombe, J.C. Interactive manipulation planning for animated characters. In Proceedings of the IEEE Eighth Pacific Conference on Computer Graphics and Applications, Hong Kong, China, 5 October 2000; pp. 417–418.
24. Kuffner, J.J., Jr.; LaValle, S.M. RRT-connect: An efficient approach to single-query path planning. In Proceedings of the IEEE International Conference on Robotics and Automation, San Francisco, CA, USA, 24–28 April 2000; Volume 2.
25. Lau, M.; Kuffner, J.J. Behavior planning for character animation. In Proceedings of the 2005 ACM SIGGRAPH/Eurographics Symposium on Computer Animation, Los Angeles, CA, USA, 29–31 July 2005; pp. 271–280.
26. Lau, M.; Kuffner, J.J. Precomputed search trees: planning for interactive goal-driven animation. In Proceedings of the 2006 ACM SIGGRAPH/Eurographics Symposium on Computer Animation, Vienna, Austria, 2–4 September 2006; Eurographics Association: Aire-la-Ville, Switzerland, 2006; pp. 299–308.
27. Esteves, C.; Arechavaleta, G.; Pettré, J.; Laumond, J.P. Animation planning for virtual characters cooperation. *ACM Trans. Gr. (TOG)* **2006**, *25*, 319–339. [[CrossRef](#)]
28. Oshita, M.; Ogiwara, Y. Sketch-based interface for crowd animation. In Proceedings of the International Symposium on Smart Graphics, Salamanca, Spain, 28–30 May 2009; Springer: Cham, Switzerland, 2009; pp. 253–262.

29. Bresenham, J.E. Algorithm for computer control of a digital plotter. *IBM Syst. J.* **1965**, *4*, 25–30. [[CrossRef](#)]
30. Pinter, M. Toward more realistic pathfinding. *Game Dev. Mag.* **2001**, *8*, 1–7.
31. Ferguson, D.; Stentz, A. Field D*: An interpolation-based path planner and replanner. In *Robotics Research*; Springer: Cham, Switzerland, 2007; pp. 239–253.
32. Nash, A.; Koenig, S.; Tovey, C. Lazy Theta*: Any-angle path planning and path length analysis in 3D. In Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence, Atlanta, GA, USA, 11–15 July 2010.
33. Faria, M.; Marín, R.; Popović, M.; Maza, I.; Viguria, A. Efficient Lazy Theta* Path Planning over a Sparse Grid to Explore Large 3D Volumes with a Multirotor UAV. *Sensors* **2019**, *19*, 174. [[CrossRef](#)]
34. Harabor, D.D.; Grastien, A. An optimal any-angle pathfinding algorithm. In Proceedings of the Twenty-Third International Conference on Automated Planning and Scheduling, Rome, Italy, 10–14 June 2013.
35. Uras, T.; Koenig, S. An empirical comparison of any-angle path-planning algorithms. In Proceedings of the Eighth Annual Symposium on Combinatorial Search, Ein Gedi, Israel, 11–13 June 2015.
36. Yap, P.; Burch, N.; Holte, R.C.; Schaeffer, J. Block A*: Database-driven search with applications in any-angle path-planning. In Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence, San Francisco, CA, USA, 7–11 August 2011.
37. Uras, T.; Koenig, S.; Hernández, C. Subgoal graphs for optimal pathfinding in eight-neighbor grids. In Proceedings of the Twenty-Third International Conference on Automated Planning and Scheduling, Rome, Italy, 10–14 June 2013.
38. Islam, F.; Nasir, J.; Malik, U.; Ayaz, Y.; Hasan, O. Rrt*-smart: Rapid convergence implementation of rrt* towards optimal solution. In Proceedings of the 2012 IEEE International Conference on Mechatronics and Automation, Chengdu, China, 5–8 August 2012; pp. 1651–1656.
39. Boardman, B.; Martínez, S.; Harden, T. Focused Refinement in the RRT*: Trading Optimality for Improved Performance. In Proceedings of the ASME 2015 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference, Boston, MA, USA, 2–5 August 2015; American Society of Mechanical Engineers: New York, NY, USA, 2015; p. V05CT08A006.
40. Geraerts, R.; Overmars, M.H. Creating high-quality paths for motion planning. *Int. J. Robot. Res.* **2007**, *26*, 845–863. [[CrossRef](#)]
41. Koyuncu, E.; Ure, N.K.; Inalhan, G. A probabilistic algorithm for mode based motion planning of agile unmanned air vehicles in complex environments. *IFAC Proc. Vol.* **2008**, *41*, 2661–2668. [[CrossRef](#)]
42. Ravankar, A.; Ravankar, A.; Kobayashi, Y.; Hoshino, Y.; Peng, C.C. Path smoothing techniques in robot navigation: State-of-the-art, current and future challenges. *Sensors* **2018**, *18*, 3170. [[CrossRef](#)]
43. Bohlin, R.; Kavraki, L.E. Path planning using lazy PRM. In Proceedings of the 2000 ICRA, Millennium Conference, IEEE International Conference on Robotics and Automation, Symposia Proceedings (Cat. No. 00CH37065), San Francisco, CA, USA, 24–28 April 2000; Volume 1, pp. 521–528.
44. Janson, L.; Schmerling, E.; Clark, A.; Pavone, M. Fast marching tree: A fast marching sampling-based method for optimal motion planning in many dimensions. *Int. J. Robot. Res.* **2015**, *34*, 883–921. [[CrossRef](#)] [[PubMed](#)]
45. Salzman, O.; Halperin, D. Asymptotically near-optimal RRT for fast, high-quality motion planning. *IEEE Trans. Robot.* **2016**, *32*, 473–483. [[CrossRef](#)]
46. Gammell, J.D.; Srinivasa, S.S.; Barfoot, T.D. Batch informed trees (BIT*): Sampling-based optimal planning via the heuristically guided search of implicit random geometric graphs. In Proceedings of the 2015 IEEE International Conference on Robotics and Automation (ICRA), Washington, DC, USA, 26–30 May 2015; pp. 3067–3074.
47. Stentz, A. Optimal and efficient path planning for partially known environments. In *Intelligent Unmanned Ground Vehicles*; Springer: Cham, Switzerland, 1997; pp. 203–220.
48. Stentz, A. The focussed D* algorithm for real-time replanning. In Proceedings of the International Joint Conference on Artificial Intelligence, 20–25 August 1995; Volume 95, pp. 1652–1659.
49. Koenig, S.; Likhachev, M.; Furcy, D. Lifelong planning A*. *Artif. Intell.* **2004**, *155*, 93–146. [[CrossRef](#)]
50. Koenig, S.; Likhachev, M. D* lite. In *Aaai/iaai*; Carnegie Mellon University: Pittsburgh, PA, USA, 2002; Volume 15.
51. Sucan, I.A.; Moll, M.; Kavraki, L.E. The open motion planning library. *IEEE Robot. Autom. Mag.* **2012**, *19*, 72–82. [[CrossRef](#)]

52. Zhang, C.; Tang, Y.; Liu, H. Late line-of-sight check and partially updating for faster any-angle path planning on grid maps. *Electron. Lett.* **2019**, *55*, 690–692. [[CrossRef](#)]
53. Pohl, I. The avoidance of (relative) catastrophe, heuristic competence, genuine dynamic weighting and computational issues in heuristic problem solving. In Proceedings of the 3rd International Joint Conference on Artificial Intelligence, Stanford, CA, USA, 20–23 August 1973; Morgan Kaufmann Publishers Inc.: San Francisco, CA, USA, 1973; pp. 12–17.
54. Dechter, R.; Pearl, J. Generalized best-first search strategies and the optimality of A*. *J. ACM (JACM)* **1985**, *32*, 505–536. [[CrossRef](#)]
55. Cohen, B.; Phillips, M.; Likhachev, M. Planning single-arm manipulations with n-arm robots. In Proceedings of the Eighth Annual Symposium on Combinatorial Search, Ein Gedi, Israel, 11–13 June 2015.
56. Botea, A.; Müller, M.; Schaeffer, J. Near optimal hierarchical path-finding. *J. Game Dev.* **2004**, *1*, 7–28.
57. Sturtevant, N.R. Benchmarks for grid-based pathfinding. *IEEE Trans. Comput. Intell. AI Games* **2012**, *4*, 144–148. [[CrossRef](#)]
58. Quigley, M.; Conley, K.; Gerkey, B.; Faust, J.; Foote, T.; Leibs, J.; Wheeler, R.; Ng, A.Y. ROS: An open-source Robot Operating System. In Proceedings of the ICRA Workshop on Open Source Software, Kobe, Japan, 17 May 2009; Volume 3, p. 5.



© 2019 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<http://creativecommons.org/licenses/by/4.0/>).