Comparison of Grid-Based Search Path Planning - Dijkstra and A* Algorithm

Authors

Kelun Liu <kelun@kth.se>

Pei-Lun Hsu <plhsu@kth.se>

(Group: Jiantong 1)

Abstract

Efficient path planning is one of the main prerequisites for fully autonomous vehicles. Especially driving in complex environments containing obstacles and street barriers is a challenging problem. In this project we implement the hybrid versions of two popular grid-based path-finding algorithms, Dijkstra and A*, and compare their performance by testing them on various maps, which simulate street environments. The result of the comparison should give readers some insights when choosing one of two algorithms to solve path-planning problems in auto-driving applications.

Keywords:Path Planning, Dijkstra algorithm, A* algorithm, self-driving, simulation.

Introduction

This section firstly offers a brief overview of this research project's background and relevant literature review, and then describes a list of research questions and hypotheses.

Literature study

There are a lot of challenges we have to overcome on the way to fully autonomous vehicles. A key step for autonomously driving is a good path planning system. For example, a vehicle navigates from a start pose to a goal pose and avoids all obstacles in an environment that does not offer any specific structure like lanes. This

case is very common during our lives such as construction sites and parking lots. While fully autonomous vehicles should be able to perceive, reason and control themselves completely, path planning systems still play a key role. Machine learning is currently one of the largest areas in artificial intelligence, but path planning will also be thought of as a necessary complement to it because machine learning decisions need to be formed autonomously based on the results of learning [7].

When path planning is processed by computers, it can only be done programmatically. Hence, the result is thus a planning algorithm that should return a set of actions that transitions the start to its goal state. All scenes, obstacles, start and goal points need to be transferred into mathematics formals. Using a matrix is a better option that can match with approximate cell decomposition, also known as grid-based. Path planning using this method can be traced back to Brooks and Lozano-Per´ez in the middle of the 80s [1]. Their basic idea is to divide the configuration space into squares with edges parallel to the axes of the space. The squares after processing will be either free, occupied or mixed according to the obstacles' configuration space, and they can correspond with elements in a matrix. The search for a path is dealt with finding a set of connected and free squares (elements) which include the start and goal configuration [1]. Jean-Claude thinks decomposing the configuration space in this method is not exact, and presents exact cell decomposition in his research [6].

In most cases, there are some differential constraints of vehicles and robots, which are inherent to the kinematics and dynamics of themself. These constraints must be taken into account during path planning, although this will not be an easy task [7]. A vehicle can get any position and orientation in a Euclidean plane. But in reality, it can move forward as well as backward, but cannot move sideways. When a vehicle would like to turn right or left, there will be a turning radius θ. This means there are fewer possible actions than degrees of freedom, which we call underactuated [8].

There are two main categories of grid-based search path planning, breadth-first and depth-first research. Breadth-First Search (BFS) was first presented by Morre and published by Lee in 1961 [5]. This algorithm can only work on graphs with equal

edge costs. From a start point, it will traverse a graph layer by layer, all points with the same depth in the graph are visited before it enters to points with the next depth. BFS is complete and looks suitable for autonomously driving, but the cost of time and memory is very high, as this kind of search is not guided [7].

Although BFS delivers optimal solutions to the discrete path planning problem, it does not consider edge cost which means it is only suitable for uniform cost graphs. Dijkstra's algorithm is put forward to solve this issue and it can be seen somewhat of a refinement of BFS. Dijkstra's algorithm divide all vertices into three sets, the closed C, the open O (design for a priority queue) and the remaining vertices. Before it works, the set C and O are empty. And then Dijkstra's algorithm begins as depicted with Appendix I. At first the start point x_s is passed to the open set O. Then the while loop (Line 5 of Appendix I) is activated, which either returns the goal point (Line 9 of Appendix II) or nothing happened in case O is also empty. When a point gets expanded, it is removed from O and moved into C. Next all edges linked with the point are calculated and also the points they linked with. If any of those points do not belong to C, the cost so far will be calculated, $g(x')=g(x)+l(x,u)$. Where $g(x)$ is the cost so far of the point x from the start point, $l(x,u)$ is the cost of the state transition from x to x' with the action u. If the resulting cost is not equal the current cost to get that point, or that point does not belong to O, the predecessor and the cost for that point will be set and the position in the priority queue will be decreased, or this will be collected by O separately. [2, 3, 7]

A* algorithm can be seen as a kind of refinement of Dijkstra algorithm to a certain extent. It uses a kind of heuristic method which allows much faster convergence under some conditions, meanwhile makes sure its optimality [4]. h(x) heuristic is the cost to come, based on an estimate of the cost from state x to the goal state x_g. A* algorithm also begins from O and C as the open and closed set separately, like Dijkstra algorithm we mentioned before. And then critical difference with Dijkstra occurs in line 18 of Appendix II, where the heuristic estimate starts to work so that the cost of a state x becomes $f(x)=g(x)+h(x)$, the priority queue will be re-ordered

according to it. It can speed up search and maintain admissibility with a standard heuristic estimate.

Research questions and hypotheses

The primary motivation of our project is to evaluate and compare, through simulations of random maps, which algorithm has better performance including distance of predicted path and execution time, for the path planning of autonomous vehicles.

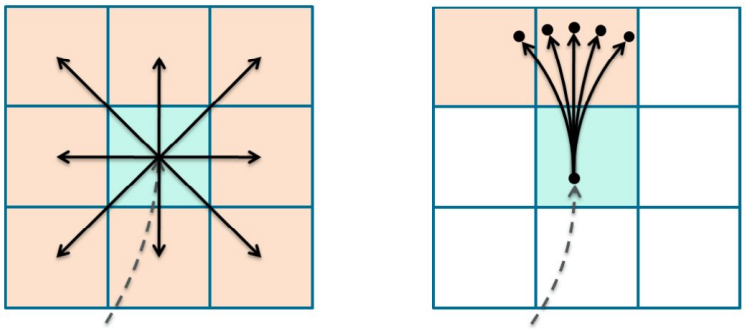
Most of the research articles we have read recently focus on two points. One is to perform mathematical analysis and comparison of algorithms without considering actual applications, such as Ortega-Arranz's book 'The shortest-path problem: analysis and comparison of methods' published in 2015. Another is to analyze the basic algorithm for a specific applied problem (such as omnidirectional robot's path planning and self-driving submarine's path planning for rescue) and then improve this algorithm facing the problem, for example Yang and etc.'s paper 'An Efficient Algorithm for Grid-Based Robotic Path Planning Based on Priority Sorting of Direction Vectors' and Masudur Rahman Al-Arif and etc.'s paper 'Comparative Study of Different Path Planning Algorithms: A Water based Rescue System'.

Our research can be seen as a collection of the two types of studies mentioned above. We set a special application problem (for self-driving cars, plan a path without guidance), then simulate, analyze and compare two basic algorithms (usually researchers will improve these two basic algorithms). we would like to find out which basic algorithm is more suitable for the path planning of self-driving cars, and provide suggestions for later researchers to choose which algorithm to improve in the initial stage.

Research Methods

Instead of conducting the comparison on the well-known traditional Dijkstra and A* algorithm [7], in this project we adopted the hybrid version of A* and Dijkstra algorithms [1]. Unlike the traditional Dijkstra and A*, which treat the map as a composition of discrete grid points and assume the agent is ideal holonomic, hybrid

algorithms consider the nonholonomic constraint of the real-world vehicle and guarantees that the path calculated by the algorithms can be actually followed by the auto-driving vehicles or vehicle like robots, which is shown in figure 1 [9].



(a) regular A* **(b)** Hybrid A*

Figure 1 - Possible neighbours of a cell. *Reprinted from Application of Hybrid A* to an autonomous mobile robot for path planning in unstructured outdoor environments, by Petereit, J., Emter, T., Frey, C. W., Kopfstedt, T., & Beutel, A, May 2012, retrieved from ROBOTIK 2012; 7th German Conference on Robotics (pp. 1-6), Copyright 2012 by VDE.*

The hybrid algorithms are the extensions of traditional Dijkstra and A* algorithm, which not only store the discrete grid states, but also store the continuous positions of the agent. By considering both discrete and continuous states of the vehicle, the algorithms can calculate the near optimal path that considering the constraints of the vehicles, e.g., maximum steering angle. The hybrid algorithms are closer to the real-world applications of path-planning/searching algorithms, such as self-driving cars and boats. Comparison of these algorithms regarding their execution times as well as their planned routes, i.e., the length of the planned path, should have more values and meet the current trend of research. The details of the implementation of hybrid A* algorithm can be found in the research related to the urban environment exploring [9]. As we can see the algorithm structures of Dijkstra and A* in the introduction section, the difference between the two algorithms is whether or not adopting the heuristic during the process of choosing the next expanding node, i.e.

the direction of the exploring. We therefore implement the hybrid version of Dijkstra by taking out the heuristic form the hybrid A*.

We run the 2 path-finding algorithms on 10 different randomly generated maps to test the performance of the algorithms on various environments, and compare their performance based on the length of the planned path and the execution time of algorithms. The 10 maps are grouped into 2 classes, round obstacles and maze-liked. The maps with round obstacles are used to test the ability of algorithms to avoid the obstacles with different size on the roads. This category of maps is also the one with less difficulty since a simple greedy search is usually sufficient to avoid the obstacles and find the path after fine parameter tuning. The maze-liked maps are used to simulate the walls and buildings along the streets, and to test the ability of algorithms to find drivable paths within the roads. It is usually more difficult to find a path in maze-liked maps, which are more similar to the real world street maps, and we need to implement a comprehensive grid-based path planning algorithms, e.g., A* or Dijkstra, to successfully solve the problems.

Results and Analysis

The results of two path planning algorithms are shown in figure 2 and figure 3, respectively.

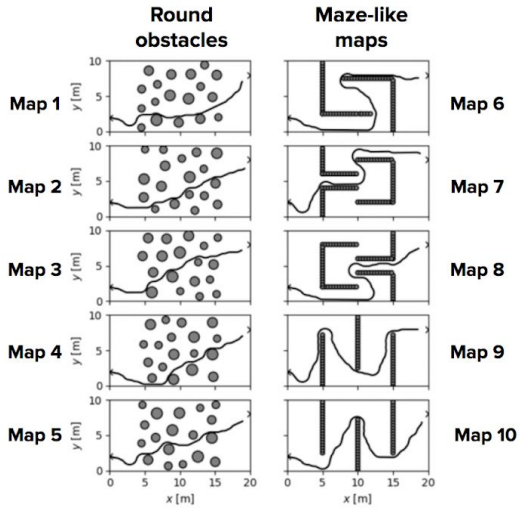


Figure 2 - Path calculated by Dijkstra

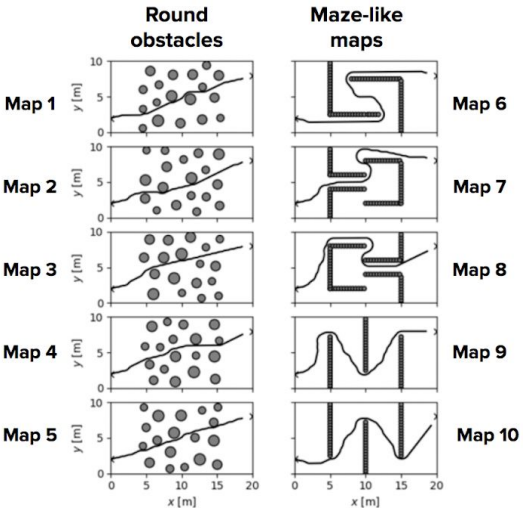


Figure 3 - Path calculated by A*

We can see from the map that both algorithms can successfully reach the goal without running outside the boundary or running into obstacles. However, the paths obtained by Dijkstra, which are shown in figure 2, are more crooked than the paths planned by the A*. This can especially be seen on the dotted maps where the algorithm should deviate as little as possible from a straight line between start and finish. The length of the paths in all maps planned by A* are shorter than those planned by Dijkstra, which are shown is figure 4. Unlike the traditional Dijkstra and A* algorithm, which generate the exactly same optimal path in the same map under same simulation setting, this result shows that when considering the constraints of the kinematics and dynamics of real world vehicle, we could effectively shorten the path from start to goal by adding a heuristic function in the planning algorithm.

Furthermore, the execution time for A* to calculate the final path is also shorter than the Dijkstra, the time gap between two algorithms exists in all testing maps, which are shown in figure 5.

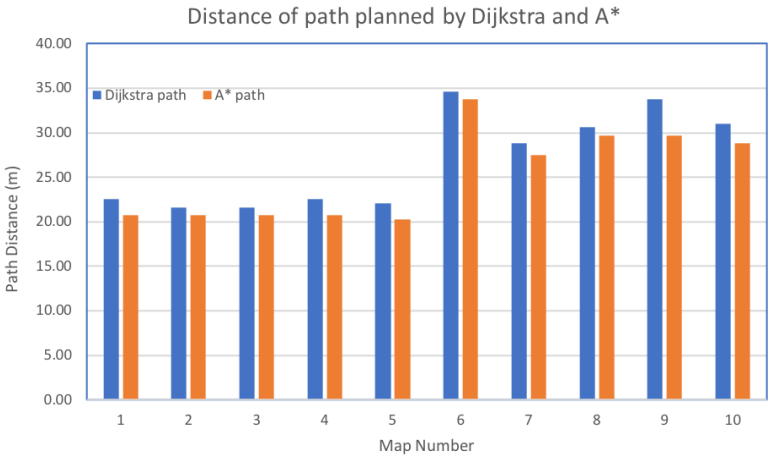


Figure 4 - Distance of path planned by Dijkstra and A*

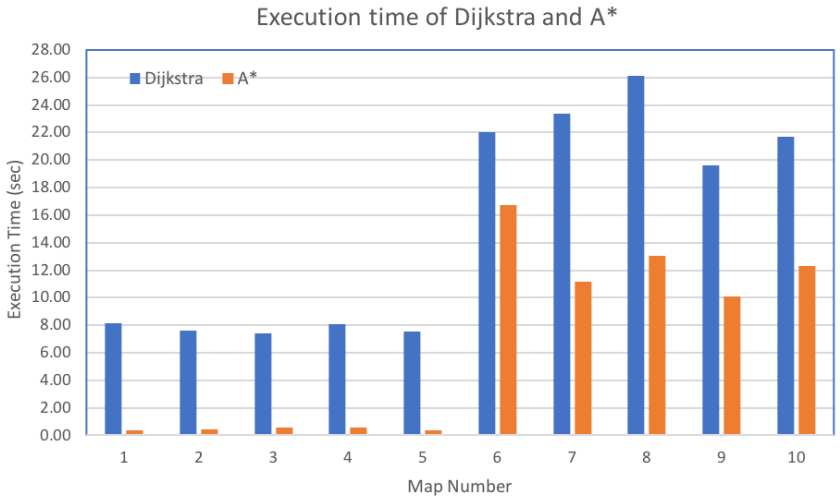


Figure 5 - Execution time of Dijkstra and A* algorithm

Discussion and Conclusion

The A* algorithm outperforms the Dijkstra algorithm both in the length of path to goal and the execution time. The result of execution time is quite reasonable since heuristic in A* can guide the searching direction toward the goal, and hence decrease the number of explored/calculated grid cells during the calculation and largely shorten the time needed to reach the goal. The improvement in speed is a well-known difference between traditional Dijkstra and A* algorithms. From our results, we can see that this improvement still exists between the hybrid version of Dijkstra and A* algorithms, which take into account the constraints of the kinematics of viehles and are relatively better in solving the real world path-planning tasks.

However, the result of the slight improvement in the path length by using hybrid A* algorithm as compared with hybrid Dijkstra is different from the behavior of traditional discrete Dijkstra and A* algorithm. As mentioned in the introduction, both algorithms update the visited cell with the shortest path from the start cell, the final paths calculated by both algorithms should be nearly same and approximate to an optimal path from start to goal. However, paths calculated by the hybrid A* are slightly

shorter than those calculated by hybrid Dijkstra in every testing map, which means that paths planned by hybrid A* are closer to the optimal path.

To explain why this happened, we have to look back into the setting of the hybrid version of algorithms. In order to discretize path-finding problems while retaining sufficient feasible path solutions, hybrid algorithms not only discretizes the positions of the agent, i.e., x and y coordinates, into grids, but also discretizes the orientations of the agent, i.e., the heading directions, in each grid [9]. This means that there could be multiple paths from start point to any grid point on the map, each path is a near optimal path with respect to its direction in the grid, but not guaranteed to be the shortest path solution. As the hybrid Dijkstra is a kind of breadth first search, these non-optimal errors could pile up while algorithm equally explore the grids in all directions in a 2D map, which is similar to a circular wavefront, and ends up a feasible but not optimal path solution. On the other hand, although hybrid version of A* has the same non-optimal characteristic, that paths will diverge from the optimal path as the grids are explored, the heuristic in A* could confine the direction of exploration towards the goal. With most of the grids explored are confined in a relatively narrow area towards the goal, hybrid A* can find a path that is more close to the shortest path than Dijkstra.

In conclusion, we implement the hybrid version of A* and Dijkstra path-finding algorithms, which take into account the constraints of real-world vehicles. We compare their performance by testing them on various maps with obstacles and barriers, which are used to simulate the complex environments on streets. The hybrid A* algorithm not only obtain the paths faster than hybrid Dijkstra, the paths planned by hybrid A* are also shorter than those planned by hybrid Dijkstra. Hybrid Dijkstra explores more unnecessary cells during the planning process, which results in longer execution time. In addition, non-optimal local solutions might pile up while Dijkstra algorithm explores unnecessary cells, which could be the cause of longer paths than A*. Since hybrid A* performs better in both execution time and path length when considering nonholonomic constraint of vehicles, one should use A* instead of Dijkstra for applications/projects in auto-driving with hybrid setting.

References

[1] Brooks, R. A., & Lozano-Perez, T. (1985). A subdivision algorithm in configuration space for findpath with rotation. *IEEE Transactions on Systems, Man, and Cybernetics*, (2), 224-233.

[2] Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to algorithms*. MIT press.

[3] Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. *Numerische mathematik*, *1*(1), 269-271.

[4] Hart, P. E., Nilsson, N. J., & Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, *4*(2), 100-107.

[5] Lee, C. Y. (1961). An algorithm for path connections and its applications. *IRE transactions on electronic computers*, (3), 346-365.

[6] Latombe, J. C. (2012). *Robot motion planning* (Vol. 124). Springer Science & Business Media.

[7] LaValle, S. M. (2006). *Planning algorithms*. Cambridge university press.

[8] Reeds, J., & Shepp, L. (1990). Optimal paths for a car that goes both forwards and backwards. *Pacific journal of mathematics*, *145*(2), 367-393.

[9] Petereit, J., Emter, T., Frey, C. W., Kopfstedt, T., & Beutel, A. (2012, May). Application of Hybrid A* to an autonomous mobile robot for path planning in unstructured outdoor environments. In *ROBOTIK 2012; 7th German Conference on Robotics* (pp. 1-6). VDE.

Appendix I

```
Require: x_s ∩ x_g ∈ X
 1: O = ∅
 2: C = ∅
 3: Pred(x_s) ← null
 4: O.push(x_s)
 5: while O ≠ ∅ do
 6:     x ← O.popMin()
 7:     C.push(x)
 8:     if x = x_g then
 9:         return x
10:     else
11:         for u ∈ U(x) do
12:             x_succ ← f(x, u)
13:             if x_succ ∉ C then
14:                 g ← g(x) + l(x, u)
15:                 if x_succ ∉ O or g < g(x_succ) then
16:                     Pred(x_succ) ← x
17:                     g(x_succ) ← g
18:                     if x_succ ∉ O then
19:                         O.push(x_succ)
20:                     else
21:                         O.decreaseKey(x_succ)
22:                     end if
23:                 end if
24:             end if
25:         end for
26:     end if
27: end while
28: return null
```

Dijkstra's Algorithm

Appendix II

```
Require: x_s ∩ x_g ∈ X
 1: O = ∅
 2: C = ∅
 3: Pred(x_s) ← null
 4: O.push(x_s)
 5: while O ≠ ∅ do
 6:     x ← O.popMin()
 7:     C.push(x)
 8:     if x = x_g then
 9:         return x
10:     else
11:         for u ∈ U(x) do
12:             x_succ ← f(x, u)
13:             if x_succ ∉ C then
14:                 g ← g(x) + l(x, u)
15:                 if x_succ ∉ O or g < g(x_succ) then
16:                     Pred(x_succ) ← x
17:                     g(x_succ) ← g
18:                     h(x_succ) ← Heuristic(x_succ, x_g)
19:                     if x_succ ∉ O then
20:                         O.push(x_succ)
21:                     else
22:                         O.decreaseKey(x_succ)
23:                     end if
24:                 end if
25:             end if
26:         end for
27:     end if
28: end while
29: return null
```

A* Algorithm