A Comparison of Path Finding Algorithms for a Self Driving Car from the perspective of an RRT* using a Kinematic Motion Model

GROUP 1:1

Anton Anderzén
1995-02-28
anderze@kth.se

Fredrik Omstedt
1997-03-16
omstedt@kth.se

Abstract

Robots that automatically can plan and execute paths are useful in many situations, such as in factories, as lawn mowers, or through self driving cars. By being autonomous, the supervision and control by humans is no longer necessary, alleviating work from people. It is important to know what type of solutions are useful when dealing with these kinds of problems, which is why solutions should be compared. In this report, a sampling algorithm, called RRT*, using a kinematic motion model is used with a car to find a path leading from start to goal trough a maze. The car then follows the found path with a simple input controller that controls the car in a Unity environment. The time performance of the car is compared to that of several other solutions, such as a deterministic path planning algorithm called A*, and a purely sensor based approach, in order to figure out why some algorithms work better than others. It is concluded that the RRT* algorithm is costly in terms of computational time when compared to the A* solutions. The benefit of the RRT* generating a traversable path, in contrast to A*, does not help due to the fact that the kinematic model cannot correctly model the advanced physics of the car. However, the RRT* solution is still a viable option, and may be more useful in environments in which grids cannot easily be applied.

1 Introduction

Automatic steering and path finding is something that is used in many applications available to the public, such as autonomous vacuum cleaners, lawn mowers and self driving cars. This enables devices that previously had to be controlled or supervised by a person to carry out tasks without the need of a human to mediate the process. In a sense, path finding helps people spend time doing more interesting things in life instead of these mundane tasks.

There are several algorithms that are used for planning paths in various environments. Two of these are A* and Rapidly-exploring Random Trees (RRT). The A* algorithm, presented in [3], can be seen as an extension of Dijkstra's algorithm (see [2] for a description of Dijkstra's). A* does not only use a mathematical discrete representation, but also takes some heuristic into account. To define this heuristic, some domain knowledge for the specific problem is required. In [3] it is proven that the path that is found by the A* algorithm is the optimal path, given the previously defined heuristic.

RRT, presented in [6], is a sampling based algorithm. When planning a path for a certain object, it samples various possible configurations for this object and try to map these together in a tree. Once it has successfully managed to sample points such that the goal node is reachable from the start node in the tree, a path has been found. RRTs are useful for problems in higher dimensions, and with nonholonomic constraints.

RRT*, presented in [5], is an extension of RRT. Compared to RRT, which converges to a solution, RRT* converges to the optimal solution by rewiring the tree during its creation.

In this paper, various algorithms (including variants of those mentioned above) will be compared when planning paths for a car in different maze environments. The algorithms will be compared on how fast the car traverses the mazes. More specifically, this paper is part of a group of papers, each presenting an algorithm that is compared with the rest. This paper presents a version of the RRT* algorithm.

It is hypothesised that the RRT* will perform better in comparison to the other algorithms tested. This is thought to be true due to it being able to cope with the nonholonomic constraints of a car, compared to A*.

It was shown that the RRT* did perform well, but not as good as the A* algorithm. This was mainly due to the random behaviour of RRT*, and because of the complexity in modeling the car. See section 4 for more information regarding these results.

1.1 Contribution

Several papers have been published that present novel solutions for path planning. Moreover, some papers have compared different variants of the same algorithm, to figure out when the algorithm performs optimally. Some of these papers will be described in section 2.

This paper contributes to this field by comparing different solutions with each other. Instead of focusing on one single algorithm or different variants of the same algorithm, many different algorithms are tested on the same problem in order to find out which one performs optimally. This is useful because it determines what direction to head in when working with similar problems, and because it shows why certain solutions are better than others. By knowing which solution is the best, one can eliminate the risk of wasting time implementing a suboptimal path planning algorithm.

1.2 Outline

Section 2 describes relevant articles and papers for similar problems. It showcases various solutions and approaches to similar topics. Section 3 describes the algorithm presented in this paper in detail; how it works and how its components were implemented. Section 4 describes the experiments conducted, and the results from these. Moreover, the results are analyzed to determine their cause. Finally, in section 5, the paper is summarized and conclusions are drawn.

2 Related work

In this section, an assortment of papers related to this one is presented. The contents and how they relate to this paper are briefly described.

In [7], an RRT algorithm with a kinematic motion model, used to plan a path for a car in a maze, is presented. It is quite similar to the problem in this paper. However, [7] compares the algorithm with one using a dynamic motion model which is more adapted to the mechanics of a car. It is concluded that the dynamic motion model generates better paths, but that it increases the computational time of the RRT algorithm.

[4] extends upon the algorithm mentioned in the previous paragraph by implementing an RRT* algorithm with a dynamic motion model. The paper showcases positive results that are applicable in real time implementations.

[9] is a comparative study of path-finding algorithms. The algorithms compared are A*, RRT, along with algorithms called Potential-field and D*. The study aims to compare these algorithms, basing the comparison on things

such as computational complexity, speed and optimality. It is found that A* is sufficient in the problems where the environment is fully known, and never changes. The D*, a dynamic version of the A*, is recommended for problems where the obstacles move or the environment is changing in some other way. The Potential-field algorithm is concluded to be a simple and effective way of avoiding obstacles. The RRT algorithm is critiqued in that it does not guarantee the optimal path. [9] focuses on the effectiveness of the entire solution, both the car controller and the path finding algorithm. It differs from this paper in that it uses the non-optimal RRT. This paper presents an RRT* that does generate an optimal path.

Another algorithm similar to both the A* and the RRT*, called Hybrid A*, is used in [8] to generate near optimal paths in an unknown outdoor environment. Hybrid A* is an extension of the A* algorithm that can work subject to constraints from the vehicle motion model, and is not limited to a discrete space. In [8], a real-time solution is presented for unknown environments. It takes into account the kinematic constraints of the vehicle and guarantees that the path is traversable.

In contrast to this paper, [10] presents a learning based approach to solving the problem of finding paths for car-like objects. A neural network is utilized for real-time collision free path planning in dynamic environments. In static environments, the paths planned are globally optimal. It is proven in [10] that the method converges.

3 Method

3.1 Configuration space

In order to successfully plan a path for the car it is necessary to know how the car can move in accordance to obstacles in the environment. The configuration space of the car defines the possible configurations the car can be in. One configuration is therefore a point in the configuration space. The dimension of the configuration space is dependent on the variables needed to define a configuration of the car. [1]

In this paper, the configuration of the car was defined as the car's position in the plane (x, y), as well as its orientation θ. This configuration was used as input to a function to determine if a collision would occur at the given position with the given orientation. As such, the car's path could be planned using points instead of the actual three dimensional car object.

3.2 RRT*

The RRT* algorithm used in this report is similar to the one defined by Sertac Karaman and Emilio Frazzoli in [5]. This section will describe the differences between the algorithm used in this paper and the one in [5]. For a more detailed description of RRT in regards to car movement, see [7].

Pseudocode describing the algorithm can be found in Algorithm 1. There are a few differences described in the functions called, these are described below. Other than that, the algorithm differs from the one in [5] in its selection of k. In this algorithm, k grows linearly with the size of the tree, it grows logarithmically in [5]. Moreover, the cost is determined using the simulated movement distance between points instead of by a straight line.

Algorithm 1 RRT*

```
1: Tree.Add($x_{start}$)
2: for $i \leftarrow 0$ to $N$ do
3:     $x_{rand} \leftarrow$ RandomPoint()
4:     $x_{near} \leftarrow$ NearestNeighbour($x_{rand}$)
5:     $x_{new} \leftarrow$ SimulateMovement($x_{rand}$, $x_{near}$)
6:     if not Collision($x_{new}$) then
7:         $X_{close} \leftarrow$ KNearestNeighbours($x_{new}$, $k$)
8:         Tree.Add($x_{new}$)
9:         $x_{min} \leftarrow x_{near}$
10:        minCost $\leftarrow$ cost($x_{min}$, $x_{new}$)
11:        for x in $X_{close}$ do
12:            if SimulateMovement($x_{new}$, x) = $x_{new}$ then
13:                if cost(x, $x_{new}$) < minCost then
14:                    $x_{min} \leftarrow$ x
15:                    minCost $\leftarrow$ cost(x, $x_{new}$)
16:        Tree.AddEdge($x_{min}$, $x_{new}$)
17: return Tree
```

3.2.1 Point sampling

The points sampled in `RandomPoint()` are sampled according to the following rules:

- 90% of the time, select a point randomly anywhere on the map where there is no obstacle.
- 10% of the time, select a point randomly anywhere on the map where there is no obstacle, closer than the currently closest point to the goal.

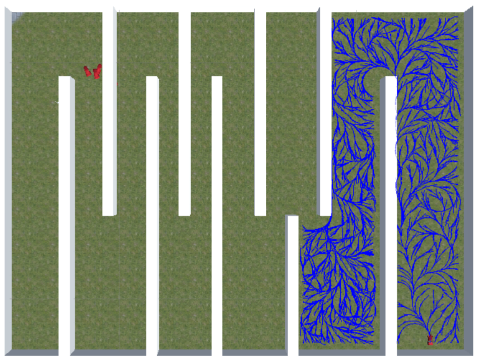

Figure 1: An example of where a Euclidean Nearest Neighbour is troublesome.

This sampling yields a certain bias to points closer to the goal, which is used to speed up the algorithm [7].

3.2.2 Nearest neighbour function

In [5], the Euclidean distances between all neighbours and the point are calculated, and the shortest distance determines the nearest neighbour. This is troublesome for several reasons. First of all, the algorithm is slow since it has to go through all points in the tree every time it looks for the nearest neighbour. Secondly, the algorithm does not take into account obstacles. The closest point could be one that is on the opposite side of an obstacle, thus eliminating the chance of finding a path between the two points.

In some cases, the above reasons makes it impossible to find a path for the car to take. This is because all sampled points that can be reached to move the path forward have nearest neighbours on the other side of an obstacle. An example of such a case can be found in Figure 1.

The algorithm used in this paper is a Pulsating Nearest Neighbour function which utilizes a grid to look for neighbours close to the point, and which stops as soon as it has found one. The pulse starts in the grid cell with the point in it. If there are several neighbours in this cell, the closest one according to Euclidean distance is chosen. If no points are found in the cell, all adjacent cells are checked simultaneously and the closest neighbour is chosen. If none of these cells contain points, all their adjacent cells are checked, and so on.

If a cell contains an obstacle, the adjacent cells of that cell are not added to the set of cells to check in the next iteration of the algorithm. This results

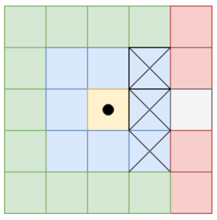
Figure 2: Showcases how the Pulsating Nearest Neighbour works in regards to the black point. The crossed out cells contain obstacles. The yellow area is checked in the first iteration of the pulse, the blue in the second, the green in the third, the red in the fourth and the gray in the fifth. The algorithm stops after the iteration where it first finds a neighbour is completed.

in the pulse being blocked by obstacles, meaning that the problem mentioned above with the Euclidean Nearest Neighbour is eliminated.

in `KNearestNeighbours(x, k)`, the algorithm works the same way but stops after it has found k neighbours.

A graphical representation of how the algorithm checks for neighbours can be found in Figure 2.

3.2.3 Motion model and controller

To be able to simulate movement between points, a motion model is required. Algorithm 1 uses the kinematic motion model presented in [7], which can be seen in Equation 1. In this equation, v is the velocity of the car, L is the length between the car's front and back wheels, and δ is the steering angle.

$$\begin{cases} \dot{x} = v \cdot cos\,\theta \\ \dot{y} = v \cdot sin\,\theta \\ \dot{\theta} = \frac{v}{L} \cdot tan\,\delta \end{cases} \tag{1}$$

A controller is used to generate input for this model when generating the path. This controller is also used to steer the car once the path has been calculated. This controller moves towards a point at all times. It has the following rules:

- δ is set to the angle to the point the car is moving towards in regards to θ, clamped to the car's maximum value.
- v is updated with constant acceleration. The acceleration is positive if the point is in front of the car, and negative otherwise. This value is clamped to the car's minimum and maximum values.
- If v is negative, δ is inverted.

- If the car is in a sharp turn (δ is larger than 80% of the maximum steering angle), the acceleration is set to 0. If $abs(v)$ is larger than 10% of the maximum speed during this case, acceleration is instead set to its negative value.

These rules are applied both when generating the path and when the car is following it. The controller also has rules that are only applied when the car is moving:

- If a sharp turn is coming up and $abs(v)^2 > 200l$, where l is the distance from the car to the turn, acceleration is set to its negative value.
- If the angle to a point p further down the path is less than $10°$ in regards to θ, if the expected orientation in p differs from θ with less than $15°$, and if there is no obstacle between the car and p, the car starts pursuing p instead.

3.2.4 Movement simulation

In `SimulateMovement(`$\mathtt{x}_a$`, `$\mathtt{x}_b$`)`, movement between the points is simulated for an amount of time steps of size 0.05 seconds, or until the point has been reached or the path collides with an obstacle. When finding $\mathbf{x}_{new}$, a maximum of 10 time steps are simulated, whereas when trying to rewire the path (line 12 in Algorithm 1), a maximum of 30 time steps are simulated.

In each time step, the controller is used to determine δ and the acceleration. Furthermore, (x, y) and θ are updated using the kinematic model and the Runge-Kutta method in regards to the time step. Finally, the velocity is updated linearly using the acceleration input and the time step.

4 Experimental results

4.1 Experimental setup

The problem was given to 13 different groups of two to three people. Each group was free to pick any way of solving said problem. In Table 1, a short categorization of the solutions chosen is made. The experiment, where each group participated, was done by each group letting their algorithm run on each of the different mazes, terrains A, B, C, D and E. No limitation in number of tries was made. The best times for each group were then entered by said group into a shared document and the logged trajectories were handed in.

Group #	Solution
Group 1	RRT*, Kinematic motion model, Point following.
Group 2	A*, Sampling waypoints.
Group 3	Hybrid A*, raycasting crash sensors.
Group 4	RRT, Kinematic motion model, Rewiring crash handling.
Group 5	Hybrid A*, Point following.
Group 6	A*, Point following.
Group 7	A*, Sensor steering & anti-crash.
Group 8	RRT*, Kinematic motion model.
Group 9	RRT, Point following.
Group 11	Dijkstra's, Point following.
Group 12	Dijkstra's, Point following.
Group 13	A*, Point following.
Group 14	RRT*, Dubin's car.

Table 1: Algorithm used by individual groups.

4.2 Experiment

In Table 2, the traversal times for each group on each terrain are presented. The algorithm presented in this paper was created by Group 1. In the following discussion we will focus on the solutions of groups 1, 2, 4, 6, 7, 12 and 14. Focusing on the groups with the better times yields more stable and comparable results, which more likely correctly corresponds to the general algorithms used, whereas the other groups' results may be biased towards their individual solutions.

Out of the RRT* groups, 1 and 14, Group 1 with the kinematic motion model achieved better results. Group 14 implemented a Dubin's car model to simulate movement. With this, Group 14 managed to find solutions to all terrains but Terrain B. The problems of Terrain B when using RRT or RRT* is that almost all points that are sampled randomly are connected to their nearest neighbour through an obstacle (as seen in Figure 1). As was mentioned in subsubsection 3.2.2, Group 1 managed to solve this problem by creating a more time efficient pulsating nearest neighbour algorithm that finds the neighbours, while taking obstacles into account.

Group 4 used regular RRT, and handled crashes by rewiring, that is, generating a new tree between the crashing point and the next node in the original tree. Group 4 mentioned at their final presentation that the solution for B took a long time to calculate, and that they used the Euclidean Nearest Neighbour function, which further showcases the importance of Group 1's alternative function.

Group 2 and 7 used A* as their main approach. Both of these groups used a heuristic that premiered driving straight ahead. Group 2 used sampling from the A* path to get waypoints. When the car drives, it turns towards

Group #	Terrain A	Terrain B	Terrain C	Terrain D	Terrain E
Group 1	28	50	8	27	31
Group 2	38	57	11	29	34
Group 3	31	216	-	-	-
Group 4	26	58	9	34	34
Group 5	60	-	23	-	-
Group 6	27	48	8	26	31
Group 7	22	42	7	22	25
Group 8	47	-	9	-	-
Group 9	240	431	13	160	161
Group 11	28	55	8	23	28
Group 12	38	80	9	33	48
Group 13	42	66	20	54	51
Group 14	48	-	11	49	54

Table 2: Resulting time in seconds on the five different terrains given. A, B and C were present from the start for groups to test their work in progress solutions on. D and E were given as a test set after the final solutions for each group were finished.

the waypoint that is furthest away and has a collision free straight path. Group 7 had outstanding results with a sensor based steering and anti-crash system. The sensor system made sure that no collisions would happen as the car would steer away from a wall if it got too close. Another key part of their success could be that they conducted experiments for braking distances and found out that it followed a linear pattern. They used this to brake just in time before a turn. Using the optimal path in terms of distance, avoiding crashes and using optimal braking gave group 7 the fastest times on all the terrains.

Group 6 had an interesting idea in implementing their A*. The only points of the path that the car tries to drive towards are the turns. This means that when the car has cleared a corner it aims directly for the next corner.

Group 12 used Dijkstra's, which essentially gives the same path as A*. However, there is no possibility to introduce a heuristic that premieres going in straight lines. This could be contributing to Group 2 and Group 6 having better times than Group 12.

The A* algorithm is easier to implement and guarantees an optimal path given a heuristic in a fixed time, compared to finding the optimal path with RRT* which is guaranteed in infinite time [3, 5]. The RRT* has the benefit of finding a path that is generated by a motion model, meaning that the path is guaranteed to be maneuverable in accordance to that model. Given a perfect model for the car used, and infinite time, RRT* would therefore find the optimal traversable path.

The controllers play an important part in making sure the car actually traverses the path as quickly as possible, but with A* always finding an optimal path in regards to a heuristic, it edges out the solution of the RRT*. This is mainly because of the complexity in finding a motion model that matches the car's movement. Even with a perfect model, the RRT* is much slower than A*. It is therefore concluded that the hypothesis presented in section 1 does not hold. For this type of problem, RRT* is not the preferred algorithm of choice.

5 Summary and Conclusions

In this report, a comparison between various path finding algorithms has been made. More specifically, the problem of finding and executing paths for a car in an obstacle filled environment was presented, and various solutions were tried and compared in regards to how quickly the car could traverse the environments. 13 groups created solutions that were tested in five different environments.

It was concluded that both sampling based algorithms such as RRT*, and deterministic algorithms such as A* successfully could find paths the car could traverse with the help of a controller. However, some solutions were better than others. The RRT* had problems with certain terrains due to its problematic way of finding paths between sample points. This could however be averted by applying a different nearest neighbour function. The RRT* was also slower and did not find as optimal paths as the A*. Due to the complexity of the car model, and the randomness in RRT*, A* was the better choice for this problem.

Even though the results showcased A* as the better algorithm, it is not certain that this is the case in general. It would be interesting to see how the algorithms compare in other types of problems, especially ones where discretization of the configuration space is not easily applied. A* is heavily dependent on this, whereas RRT* is not. In such a problem, given a good model, RRT* might be the preferred algorithm. Further research must be conducted in order to determine if this is the case.

References

[1] Wolfram; Hutchinson Seth; Arkin Ronald C.; Kantor George A.; Kavraki Lydia E.; Lynch Kevin M.; Thrun Sebastian Choset,

Howie; Burgard. *Principles of Robot Motion : Theory, Algorithms, and Implementations*. MIT Press, Cambridge, 2005.

[2] Edsger W Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

[3] Peter E Hart, Nils J Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[4] Jeong hwan Jeon, Raghvendra V Cowlagi, Steven C Peters, Sertac Karaman, Emilio Frazzoli, Panagiotis Tsiotras, and Karl Iagnemma. Optimal motion planning with the half-car dynamical model for autonomous high-speed driving. In *American Control Conference (ACC), 2013*, pages 188–193. IEEE, 2013.

[5] Sertac Karaman and Emilio Frazzoli. Sampling-based algorithms for optimal motion planning. *The international journal of robotics research*, 30(7):846–894, 2011.

[6] Steven M LaValle. Rapidly-exploring random trees: A new tool for path planning. 1998.

[7] Romain Pepy, Alain Lambert, and Hugues Mounier. Path planning using a dynamic vehicle model. In *Information and Communication Technologies, 2006. ICTTA'06. 2nd*, volume 1, pages 781–786. IEEE, 2006.

[8] J. Petereit, T. Emter, C. W. Frey, T. Kopfstedt, and A. Beutel. Application of hybrid a* to an autonomous mobile robot for path planning in unstructured outdoor environments. In *ROBOTIK 2012; 7th German Conference on Robotics*, pages 1–6, May 2012.

[9] Gopikrishnan Sasi Kumar, BVS Shravan, H Gole, P Barve, and L Ravikumar. Path planning algorithms: A comparative study. 12 2011.

[10] Xianyi Yang and M Meng. An efficient neural network model for path planning of car-like robots in dynamic environment. In *Electrical and Computer Engineering, 1999 IEEE Canadian Conference on*, volume 3, pages 1374–1379. IEEE, 1999.