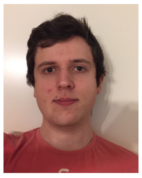
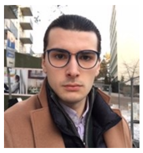
Report template for the project in the course DD2380 at KTH

GROUP1:2

Marcus Andersson
1994 06 23
and8@kth.se

Marko Lazic
1995 01 17
mlazic@kth.se





Abstract

The purpose of this paper is to present research conducted on enabling a simulated car to travel from point A to B, as fast as possible. There were two problems to be solved, the first one included defining a C-space for a path to go through, and also creating the path itself. The second problem was getting the car to travel as fast as possible, without over-steering and crashing. A literature study was initially made to evaluate existing technologies that solves this problem. After that, experimentation was made, while making changes to some popular existing techniques for path-finding. We found out that it is possible to use a discrete path, by using the Hybrid-A* algorithm to solve this problem. Despite the fact that the simulated car uses continuous movement. To make this possible, our solution required the use of a PID controller. Using the A* algorithm in conjunction with a PID-controller seems to be feasible solution. The only problematic aspect turned out to be finding the right parameters to use as input. These heavily depend on how the physics in the program are simulated. To test the solution, we have used a tool called Unity. There we created an AI for which a car had to speed through a map with obstacles. This paper can be used by game developers who want's a fast and easy way to move cars.

1 Introduction

Initially a literature study was conducted on existing solutions. We studied path-finding algorithms such as A*, Hybrid-A*, RRT and RRT*. Different ways to aid algorithms, like Voronoi diagrams, obstacle distances for car-like robots was considered. We saw this as important to know about, since we wanted to start out from the very top of modern path-finding technology, and then improve it for this specific issue. By evaluating existing algorithms strengths and weaknesses, one can make informed judgments on what aspects to keep in a solution, and potential dangers to avoid. One should avoid potential problems that can hamper solutions from being success-full. For instance, RRT is efficient for finding an optimal path, but it's found paths are usually not in forms of a straight line. Cars must travel periodically in straight lines to achieve the best speed possible. Thus, we were made aware that RRT has a high probability of causing issues.

Certain things were already known to the researchers before the work began. There exists some competitive racing games which requires players to reach a goal on a track as fast as possible. At least one researcher has tested some of these games and gained an intuition for what lines a car should follow on a track to maintain a good momentum. Similarly like a professional racing driver can tell whether a position on a road, given a certain velocity, is good or bad. Furthermore, courses taken at KTH about algorithms in general made it clear that it is not feasible to find and test all possible paths. That would take too much time. The way to solve this problem must therefore be in the form of an approximate solution.

This study will show that it is possible to solve an advanced problem relatively easily. While it may not be an optimal solution, it may be good enough for the video game industry. In video games, it is usually not wanted to have an AI that is always better than the human players, otherwise it would not be possible to win the game. That said, our solution has unknown potential. There was not enough time to investigate all the possibilities that our implementation allowed for. By tweaking certain values in the code, different kinds of paths can be achieved. The importance of path-finding algorithms is not limited to video games. It is used for GPS systems as well as self-driving cars. This solution should not be used for self-driving cars, but it may be used for navigation using GPS. The rationale behind this claim is that our tests sometimes showed that crashes occur.

This paper bridges the gaps that exist today between theory and implementation. There is an abundance of pseudo-code for algorithms. Pseudo-code can give an estimate for how efficient code will be. However, diverse scenarios and use cases will test the productivity in different ways. We have by making experiments proved how some of these theoretical aspects can be implemented in a feasible way. Note that there are many ways to implement theoretical knowledge poorly in programs, resulting in negative outcomes.

1.1 Contribution

This paper enables programmers to learn how to use the Hybrid-A* algorithm and a PID controller. By seeing this as an example, individuals can see generally how to go about when creating AIs for cars in Unity. This should in practice also benefit anyone who is creating a game involving continuous movement and collisions. Apart from existing knowledge on Hybrid-A*, we show an unique way of defining costs when finding a path of minimal cost possible.

1.2 Outline

In Section 2 we bring up other research papers that were of use, or considered through out our work. Some of these papers had a large influence on how our implementation took form. Section 3 describes in detail how the implementation looks like. How paths are found and velocities controlled are explained. Section 4 shows results from experiments that were conducted. The tests consisted of checking how fast the car could travel between two points on different maps with obstacles. Section 5 sums up the insights gained from this project. Potential improvements and future work is also mentioned.

2 Related work

The Hybrid-A* path-finding algorithm is a modern way to find the shortest path between two points. We found out that other researchers that made it hybrid did not allow the car to steer in just any direction [5]. The paper by Janko Petereit and his colleges, takes into account the kinematic limitations of the object using a path. Each time a path is calculated, turning happens slowly and not instantly. Their C-space consists of three dimensions, one is for the plane by which a robot can travel, and an extra dimension is for the angle of the robot. We copied this idea. Depending on how the car is oriented, only certain future points can be reached given a current position. This helped to resolved a problem where the path would make steep turns. They also propose a way to extend the Hybrid-A* algorithm to find a set of way-points instead of just one goal. Another paper about Hybrid-A* importantly pointed out the idea of associating points in a grid with different costs [3]. They mention a way to use the Voronoi field to calculate the cost grid. To make optimizations, they also delay using the Hybrid-A* algorithm. They take the Voronoi path and make coordinates that allow for proper steering less expensive. Also, interpolation is performed using gradients. This inspired us to do so that points that are close to walls get more expensive than normal. As a result, our algorithm will not choose paths that are lined up with walls. If such paths were chosen, collisions would be more likely to occur, and navigating through turns would become sub-optimal.

Some famous techniques were considered for making the car travel safely. One technique is by using a Voronoi diagram [1]. We did not use this, since it may cause the car to choose an unnecessarily safe route to a goal, loosing valuable time in the process. In theory however, Voronoi diagrams could help when defining a cost-grid such that it becomes easy to tweak the risk contra speed ratio when finding a path. F.Aurenhammer and H.Edelsbrunner wrote a paper about how to implement an algorithm that creates a Voronoi diagram. This is for when using a plane. It is seen from research papers that Voronoi diagrams is a popular method to use.

It is possible to take out parts of our implementation and switch them for other algorithms that solve the same problem. While RRT was avoided, it is an upcoming algorithm being researched quite a lot [4]. RRT decides on new points to take randomly. The paper on "Informed RRT*" by Jonathan D. Gammell and peers, mentions how to improve RRT*. By making changes to how data is sampled, the algorithm can become more efficient. Their tests showed that Informed RRT* is substantially faster, when compared to

RRT*. Our A* implementation could without much effort, be swapped for RRT, while keeping the hybrid functionality, such as the cost grid. If RRT or some other algorithm outperforms A* in tests, it may be worth considering trying out these as well. Initial conditions and use cases will determine what algorithm is the most suitable. To decide on what input to use for the car, it was necessary to find out about how a car behaves in terms of physics. In our implementation, the physics system was provided by Unity and also approximated. How to simulate a cars behaviour mathematically has been researched for a long time. The paper on trajectory from 1997 by Adam W. Divelbiss and John T. Wen [2] describes models to use when predicting a cars movements. This information is important when setting up a PID controller, like the one we used. Since you need to make predictions of where the car is going to end up when steering, accelerating or breaking. The reason for this is so that the car can, within some error interval, follow the path that was laid out for it.

3 My method

3.1 Hybrid A*

The Hybrid-A* method used a 2D integer array as a C-space. Indexes marked with a 0 was defined as traversable, while a 1 meant that there was an obstacle. Rows represented x-coordinates in a given map, and columns y-positions. There are both benefits and caveats by making the C-space discrete. Having a large map with many indexes can make A* slow. A* requires a discrete space to work. In a similar way, a cost-grid used by the Hybrid-A* was also made using a 2D integer array. The cost-grid had the same amount of rows and columns as the C-space. When A* looks for a shortest path, it generates lists of nodes representing paths. The first path found is assumed to be the least expensive one. This is because each path generated is saved to a priority queue, and the one with the least cost is always picked when trying out new nodes. In addition to just generating nodes with coordinates, the planned orientation for the car is stored as well. The orientation in a node could be set to be one of 8 discrete orientations. Since each square in a grid has 8 adjacent squares (unless it is on the edge). To limit the possible changes a path can take in terms of turns, the current orientation in the last node in a path was taken into consideration. By doing this we eliminated the possibility for the path to move sideways, in relation to the cars planned orientation.

Before the program started, the values for the arrays were calculated. Then they could be used in real-time to find paths relatively quickly. Our implementation avoided the fact that paths can be too close to obstacles in a rather safe manner. When the C-space was calculated, obstacles were made thicker. This way, it became impossible for the A* algorithm to pick a path that the car couldn't use. Just adding any thickness was not a trivial solution. In some maps, the start or goal coordinates were close to the walls. If too much obstacles were added to the C-space, the A* would not be able to find any path at all. Therefore, the code iteratively tested if certain amounts of thickness could be added, without making it impossible for A* to find a path. The cost-grid was used by A* to help decide on how expensive is was to travel between coordinates. We made grids close to the walls more expensive then and the ones close to the edges extra expensive. In addition, changing the current direction added an extra cost to the path being generated. An optimal path is by A* defined as the least expensive path. To prevent immediate turns, it was made impossible to make two turns in a row. Once A* had decided on changing direction, a counter was added so that no new direction could be picked as long as the counter had a value larger than 0.

By adding extra cost when changing direction and when approaching walls we wanted to get the path that have as little turns as possible, so that it can be traversed fast. At the same time we wanted that it to be safe by making it as far away from walls as possible. In that sense the result of the hybrid A* generates a naive, simplified version of the Voronoi path.

3.1.1 Path following

The resulting path obtained by A* does not represent the optimal racing path. Therefore it's not followed precisely, but is rather used as a reference for creating a final path. This includes using two different heuristics. The first heuristic generates the rough first version of the final path. It works by iterating trough the solution from A*, and finding the node that is the furthest away from current position and that is still reachable by the car. By reachable we mean a straight line without obstacles in-between. That node is then added to the final path, together with the 5 nodes that come before and 5 nodes that come after. If they are not part of a straight line they are added. If they are part of a line, the node furthest away is added. The current position is then updated to the last added node and the process is repeated until the final node is reached. The main reason for doing this is because we want the final path to follow the path from A* as closely as possible on some critical points. In particular, we want to use the original path for corners and points close to the obstacles. However, we prefer more straight paths where that is possible. Since this strategy will not yield the optimal path either, another heuristic is used to try to compensate for that while driving the car.
Because of the method above where where 10 additional nodes are added for almost every node in the final path, the nodes tend to come in batches of 11 nodes. Some of these batches will only make the final path slower by adding more turns then necessary. Thus, after reaching the end of the current batch of nodes we check if the next batch can be completely ignored. If so just continue to the reference point that comes after the ignored batch. This is done by checking for obstacles between the current position of the car and the reference point that comes after.
Steering of the car is controlled by the simple P controller with the setpoint variable being defined as the angle towards the next reference point, and the measured process variable as the current angle of the car. The variable being controlled is the steering input to the car.

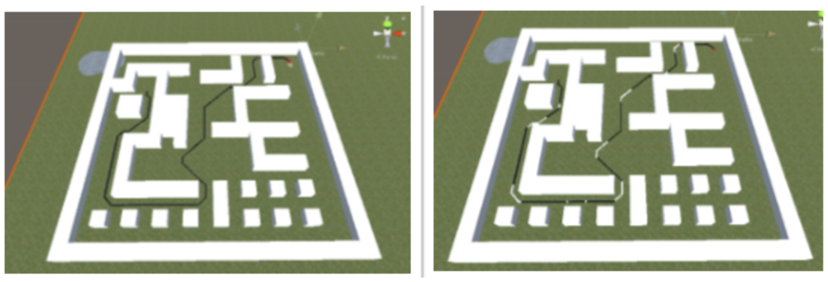


Figure 1: Final path before the heuristic and after. Nodes includes in the final path after applying the heuristic are colored white.

3.2 Velocity

Velocity is controlled by the PI controller with the setpoint variable. It's defined as the target velocity. The measured process variable is set as the current velocity of the car. Target velocity is predefined at each node in the final path during the path generation. At each batch of nodes the incoming and the outgoing car angles are calculated. Then the difference between them is compared to something that we define as maximum manageable angle. Maximum manageable angle is the angle that is experimentally determined as the largest corner angle that we can handle while driving the car with a minimum corner velocity (preset to 3 units per second). How close the corner angle is to the maximum manageable determines how close we must be to the minimum corner velocity. How much and when we need to brake is determined only by how close we are to a corner. The closer we are to the corner the more we need to brake if the current velocity is greater that the targeted one.

4 Experimental results

The code was applied to five different obstacle courses and the best times and trajectories were saved. The Hybrid-A* managed to find a traversable path for all courses.

Figure 3 contains the best results for all groups and out results are visible in row marked with G2 (group 2). The results showed that our algorithm can find a solution for all obstacle courses in satisfactory time. One characteristic of our result is that the car never crashed on any of the courses. That was in a way our goal because we made an assumption that crashes are always more expensive in the terms of time then adapting speed in order to avoid them completely. The result from the group 7 showed that our assumption was wrong and that driving fast can compensate even for eventual collisions. Another flaw in our approach and the reason why group 7 is significantly faster then us despite the fact that our A* solution is very similar is that we only take the angle of the corner into the consideration when calculating the target velocity and not the maneuver space that we have. Figure 2 shows Hybrid-A* solution for terrain B and C. Our motion controller does not have any way of differentiating the corners in these two terrains. This becomes problematic when determining the maximum corner speed since in reality, that is largely affected by the maneuver space that we have. The fact that we can not completely adapt our speed to varying situations, and the fact that we made an assumption about the cost of crashing, made us make our motion controller follow the A* solution exactly in corners. This come with a huge performance penalty, especially on terrains that have sharp corners. This is affirmed by looking into the Figure 3 where we can see that the difference between out best solution and the solution of the group 7 is greater on terrain A than on terrain B, despite the fact that the path on terrain B is significantly larger.

It is sometimes faster to take a longer path trough a corner if that will result in the larger outgoing velocity. That is also well demonstrated by the group 7 as they sometimes drive further away from a obstacle if the turn is sharp and also adapt the speed to the maximum turning radius that they can make. One final and potentially critical difference between our solution and the solution of the group 7 is using the handbrake and calculating the last possible moment to brake in order to accomplish desired corner speed. This pretty much guarantees that the straight between two corners will be driven at the largest possible speed. Our approach of always breaking at some distance from the corner no matter what the difference between the

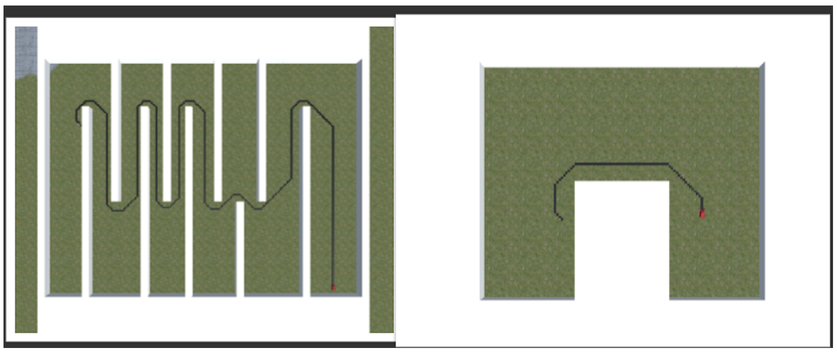
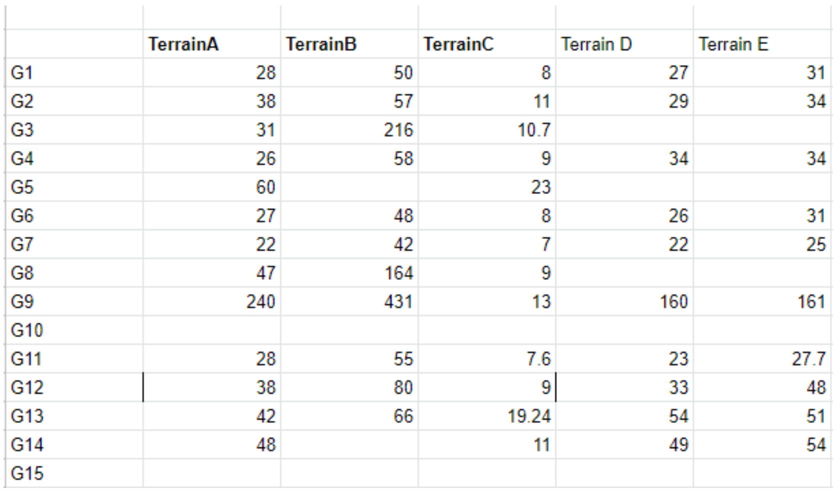


Figure 2: Hybrid-A* paths for terrain B and terrain C

current and the desired velocity is seems to be safer but not as fast solution.

Group 6 had a very similar solution to ours with the Hybrid-A* and a waypoint based motion controller. Figure 3 shows that they have beaten us on all different terrains. However, the difference is significantly higher on terrains A and B than D and E. One possible explanation to this is that their solution was a bit biased towards the training terrains (A,B and C), and that they have relied on their crash handler to do the job on the unknown tracks. Comparing their best solution on the terrain E with ours gives the impression that they are making a lot of last moment sharp turns in order to avoid direct crash into the obstacle. Having this functionality allows them to drive faster. On the other hand our path is smoother, without sharp and sudden turns, but not as fast.

Analyzing the result from the other groups showed that four of the fastest five groups used some kind of Hybrid-A* solution. The reason why this solution is dominant opposed to the RRT (and RRT*) can be found in several reasons. The first reason lies in the fact that these are probabilistic algorithms and that their solution can vary a lot. In theory RRT* will find the optimal solution as the time goes to infinity, but in practice that is not feasible. The biggest reason can probably be found in the imperfections of the motion model and the uncertainty about the Unity car psychics in general. The difference between the simulated and the exact position of the car will almost certainly lead to crashes which are time consuming. However, A* always finds the same solution so there in no uncertainty. One problem with our Hybrid-A* solution is that it does not fully consider the movement constraints of the car. This could be particularly problematic in very tight areas, where it may even fail to find any path.



	TerrainA	TerrainB	TerrainC	Terrain D	Terrain E
G1	28	50	8	27	31
G2	38	57	11	29	34
G3	31	216	10.7		
G4	26	58	9	34	34
G5	60		23		
G6	27	48	8	26	31
G7	22	42	7	22	25
G8	47	164	9		
G9	240	431	13	160	161
G10					
G11	28	55	7.6	23	27.7
G12	38	80	9	33	48
G13	42	66	19.24	54	51
G14	48		11	49	54
G15					

Figure 3: Table with the best results for respective groups

The explanation to why the groups 6 and 7 had the fastest car can be found in the simplicity of their solution, as well as the simplicity of the problem itself. All of the problem instances were quite simple in the sense that even a naive Hybrid-A* implementation with a car that only moves forward could find a traversable path. The core of this problem was not how to find a path but rather how to follow it quickly. Group 7 had a solution consisting of always accelerating if the speed was below 15. As well as waiting to break until the last possible moment, and passing the corners as fast as possible. Their solution showed to produce the best results. They also made use of sensors, that prevented the car from hitting the walls.

5 Summary and Conclusions

Our Hybrid-A* solution with the simple motion controller have managed to handle all of the obstacle courses within a reasonable amount of time. However there is sill a lot of room for improvements, especially in the area of the motion controller algorithm. There is also experimentation that could be made by inserting varying combinations of values to the heuristics used. Different inputs will produce different types of paths. Looking into the other results and implementations our conclusion is that Hybrid-A* is better suited for this particular problem than the RRT* algorithm.

References

[1] Franz Aurenhammer and Herbert Edelsbrunner. An optimal algorithm for constructing the weighted voronoi diagram in the plane. *Pattern Recognition*, 17(2):251–257, 1984.

[2] Adam W Divelbiss and John T Wen. Trajectory tracking control of a car-trailer system. *IEEE Transactions on Control systems technology*, 5(3):269–278, 1997.

[3] Dmitri Dolgov, Sebastian Thrun, Michael Montemerlo, and James Diebel. Practical search techniques in path planning for autonomous driving. *Ann Arbor*, 1001(48105):18–80, 2008.

[4] Jonathan D Gammell, Siddhartha S Srinivasa, and Timothy D Barfoot. Informed rrt*: Optimal sampling-based path planning focused via direct sampling of an admissible ellipsoidal heuristic. *arXiv preprint arXiv:1404.2334*, 2014.

[5] Janko Petereit, Thomas Emter, Christian W Frey, Thomas Kopfstedt, and Andreas Beutel. Application of hybrid a* to an autonomous mobile robot for path planning in unstructured outdoor environments. In *Robotics; Proceedings of ROBOTIK 2012; 7th German Conference on*, pages 1–6. VDE, 2012.