Robotic Path Planning

Rémy Hidra

August 30, 2020

1 Introduction

In the field of robotics, Unmanned Aerial Vehicles (UAV), especially multi-rotor copters, are increasingly important in indoor navigation scenarios. For applications like emergency rescue, product delivery or warehouse management, we have a strong need of developing efficient and reliable navigation algorithms for any kind of hardware. Small quadrocopters are widely used in this area, because they are cheap, convenient to use and can be found in any sizes.

This report presents the beginning of a research work to develop an autonomous UAV autopilot software. More specifically, in a first section, we introduce a few interesting path planning algorithms we decided to compare. In a second section, we present the setup we choose for our robotic experiments. Then, in a third and fourth section, we show how we solved key issues in the navigation pipeline. Finally, we present a few experiments in order to compare each path planning algorithm in multiple scenarios.

2 Path Planning Algorithms

In a navigation pipeline, the path planning algorithm has a critical role. Using only a start and goal points, and a potentially incomplete map of the environment, it is supposed to output a path linking the two points, without going through any obstacle. Preferably, the path needs to have some properties. It should be as optimal and short as possible, but also applicable to the robot. For instance, with a rover robot, we may want to avoid doing too many rotations, even at the cost of the path becoming longer.

2.1 A*

A* is a famous graph path search algorithm, widely used in robotics, video games and most path planning applications. It is an improvement of the Dijkstra algorithm and has many variants and small optimizations, which makes it very adaptable to every situation. A* core components are the heuristic h and the cost function g. Together, they are used to give a rank to each node n, with $f(n) = g(n) + h(n)$. The heuristic represents a potential cost of a node in

relation to the goal. The ideal $h(n)$ is the true distance to the goal using an optimal path. The function can be chosen depending on the requirements of the application. Therefore, with the right heuristic, A* can be optimal. However, in most cases, it will be sub-optimal, but still good enough for our application. Moreover, we can compensate sub-optimal paths using path optimizations.

Because A* is mainly a graph path search algorithm, it only works using a graph. However, in robotics, we are often working in a continuous environment. In other words, a critical part of the A* algorithm is also to discretize the map in a graph efficiently. If we discretize the environment using a regular grid, a granularity of the grid too high may make the algorithm not computationally efficient, while a granularity too low may not found a path and get stuck in narrow passages. Some optimization, using Voronoi graphs might be useful to build an efficient and adaptable grid.

2.2 RRT*

Rapidly-exploring Random Tree (RRT) is a class of sampling based path planning algorithms, designed to work in continuous environments. RRT* is an optimized version which aims to create a tree, beginning at the starting point, and going to most points in the map. During each iteration, it samples an random point x_{rand} in the space and computes the closest node in the tree x_{near}. Then, it tries to add a new point x_{new} connected to x_{near} at a specific distance d and in the direction of x_{rand}. If x_{new} is a valid point (i.e. does not overlap with an obstacle), it is added to the tree. In the RRT* variant, the tree can be rerouted, in case using some neighbors node improves the quality of the tree.

On the contrary of A*, RRT* is made to work in continuous environments, which makes it ideal for robotic applications. It is statistically optimal. In other words, if we run the algorithm for an infinite amount of time, so that it can sample every point of the environment, the path produced is optimal. Practically, it is obviously impossible to do. Thus, we just run the algorithm for at least a minimum number of iteration i_{min} before stopping. Then, if the goal is still not part of the tree, we continue to iterate until a valid path is found. If no path can be found, we stop the algorithm when we reach i_{max} Therefore, contrarily to A*, we do not necessarily stop when a valid path is found, and continue to iterate in case a future version of the tree improves the path to the goal. Because the algorithm is based on randomness, the path outputted is not always the same, especially when the algorithm runs for a limited number of iterations.

2.3 Path Optimizations

When using sub-optimal path planning algorithms, some simple post processing techniques can be applied in order to improve the quality of the path. Especially, with RRT* with a few iterations, the random nature of the tree creates a lot of unnecessary turns and sharp angles. To solve this problem, we introduce two

very simple algorithms. One is an **over-sampling** algorithm, which increases the number of node in the path, in order to always have nodes separated by at most a distance d. Then, we use a **filtering** algorithm, to remove unnecessary nodes. If a node further in the path can be directly connected to a node earlier, the nodes in between are discarded and both nodes are connected. This simple algorithm is very efficient to fix the sub-optimality of RRT* and approach the optimal path as much as possible. It can also be useful to remove the structure of the path found by A*, which are constrained by the grid, which creates unnecessary turns to reproduce a diagonal straight line. In our work, we will compare the A* algorithm, the RRT* algorithm and an optimized RRT*, using an over-sampling and a filtering algorithms.

3 PX4 ROS Toolchain Setup

ROS is a popular robotic tool used in most setup. It can be seamlessly used in a simulation or a real-world work. We couple it with **PX4**, an autopilot firmware running in the flight controller unit (FCU) of a UAV. The FCU is the most important element in a UAV. It is in charge of transmitting instructions to rotors and receiving information from IMU sensors. Using a control law, the FCU can let the drone hover over a specific area or let it be controlled, either by a user with a remote controller or an external software.

ROS is middle-ware used by the external autopilot to communicate with the firmware. It standardise the communication protocol, with low level hardware, and can also be used by higher level components, for instance to connect a camera to a computer vision software. It also works as a distributed system, which makes it adaptable to fragmentation in multiple computers, and resistant against hardware failure. The communication protocol used to communicate between PX4 and ROS is the **MAVLink** protocol. A node called **MAVROS** is created and MAVLink messages from and to PX4 are published to MAVROS topics. In other words, we can build a simple ROS node which will communicate with MAVROS in order to move the UAV and get data from sensors. Lastly, thanks to the ROS middle-ware we can easily use our setup in a simulated environment. We will use the Gazebo simulator, which has a good compatibility with PX4. In a real-world scenario, we would probably have a separate on-board computer on the UAV with our ROS node running, with the role of controlling the UAV. We are running the entire ROS, PX4, MAVROS and Gazebo toolchain in Ubuntu 18.04.4 LTS.

4 Mapping the Environment

4.1 Introduction to the Octomap

A major problem in navigation is the mapping. In other word, we need an efficient way to know the distribution of the obstacles in our environment. This representation has to be flexible enough, to be used by a robot made of sensors

which capture an imperfect representation of the world. A classic solution in robotic is to use an **octomap**. An octomap is a tree based representation of an environment occupation, where the resolution of the tree can be flexible and expanded as needed for the application. Each node in the tree corresponds to a specific 3D cube in the world and can tell if the cube is *occupied* or not. If the cube is too big and we need a better resolution, we can expand the node in the tree and each of the eight children the node correspond to the eight sub-cube of the parent cube. If a leaf node is occupied, all of its parents are occupied. A node can be either *occupied*, *free* or *unknown* (e.g. if the sensors never sensed this area). Many implementations are already widely used in the ROS ecosystem. Practically, we just add an octomap ROS server to our system. Provided the reference frames are setup correctly, the server will handle point cloud updates from sensors and octomap serving to applications. Similarly, different libraries can be used in a program to efficiently read an octomap object. Because of all theses reasons, we will work with octomaps in our implementation.

4.2 Octomap Ground Truth

In our project, we do not want to bother with incomplete information from sensors. In other words, we want to instantly execute the path planning on the ground truth world map. This should not be an issue, however, it is not necessarily a common application. Thus almost no resources exist to obtain the ground truth environment from a Gazebo simulated world. To solve this problem, we built a simple pipeline, in order to convert a Gazebo world to an octomap binary.

First, we parse the `.world` file and extract information about object position and orientation. Each object is represented by a model `.sdf` file, which links a collision model as simple geometric shapes or in a Collada `.dae` file. From this data, we recreate another `.dae` file, which represents the entire Gazebo physical world. Then, we can easily convert the `.dae` file to a `.obj` with Blender and to an octomap binary with the `binvox` and `binvox2bt` program. This file can be easily loaded to the octomap server ROS node, as the ground truth data.

5 Flying the UAV

In order to fly the UAV, our software needs to communicate with the PX4 firmware using the MAVLink protocol, which is encapsulated in the ROS ecosystem in the MAVROS node. The usual system for indoor UAV autopilot with our toolchain is to use the PX4 **Offboard** mode. It is a mode where no user remote controller can communicate with the FCU. Instead, the firmware listens for MAVLink Offboard messages. These can indicate either a specific position and orientation to attain, or they can contain more low level information, like velocity and acceleration data. If we are just transmitting the desired position, the firmware is automatically deducing the velocity to apply to each rotors

depending on the distance to the waypoint. In order to move to a waypoint relatively slow, we need to separate each waypoint by a relatively low distance. We can already see the usefulness of the *over-sampling* path post-processing function in order to easily regulate the velocity of the UAV along the path.

6 Path Planning Benchmark

In order to move forward in developing our autopilot, we need to choose a path planning algorithm to implement. This algorithm needs to be specific enough to be usable for our robotic application, but also generally efficient enough to be used in many different situations. It should also be computationally light, to be able to run on a small on-board computer in real-time, in order to face dynamic environment modifications as fast as possible.

6.1 Experimental setup

We decided to compare two path planning algorithms, A* and RRT*. Since RRT* with a few iterations does not generally produce paths of a good enough quality, we also compare them to an RRT* path with over-sampling and filtering path optimizations. We will refer to this algorithm as **RRT* Optimized**.

We run the entire toolchain described earlier, including ROS, PX4, MAVROS, Gazebo, Rviz and our software on the same laptop. **Rviz** is used for visualization and diagnostic. No visualization is done during the core path computation. We use Ubuntu 18.04.4 LTS on a 64 bits CPU Intel Core i7-4702MQ at 2.20 GHz with 16 Gb RAM.

We imagined six tests, which all happen in the same world, labelled from *A* to *F*. Each test has its own difficulties for each algorithm. We present all the tests in figure 1.

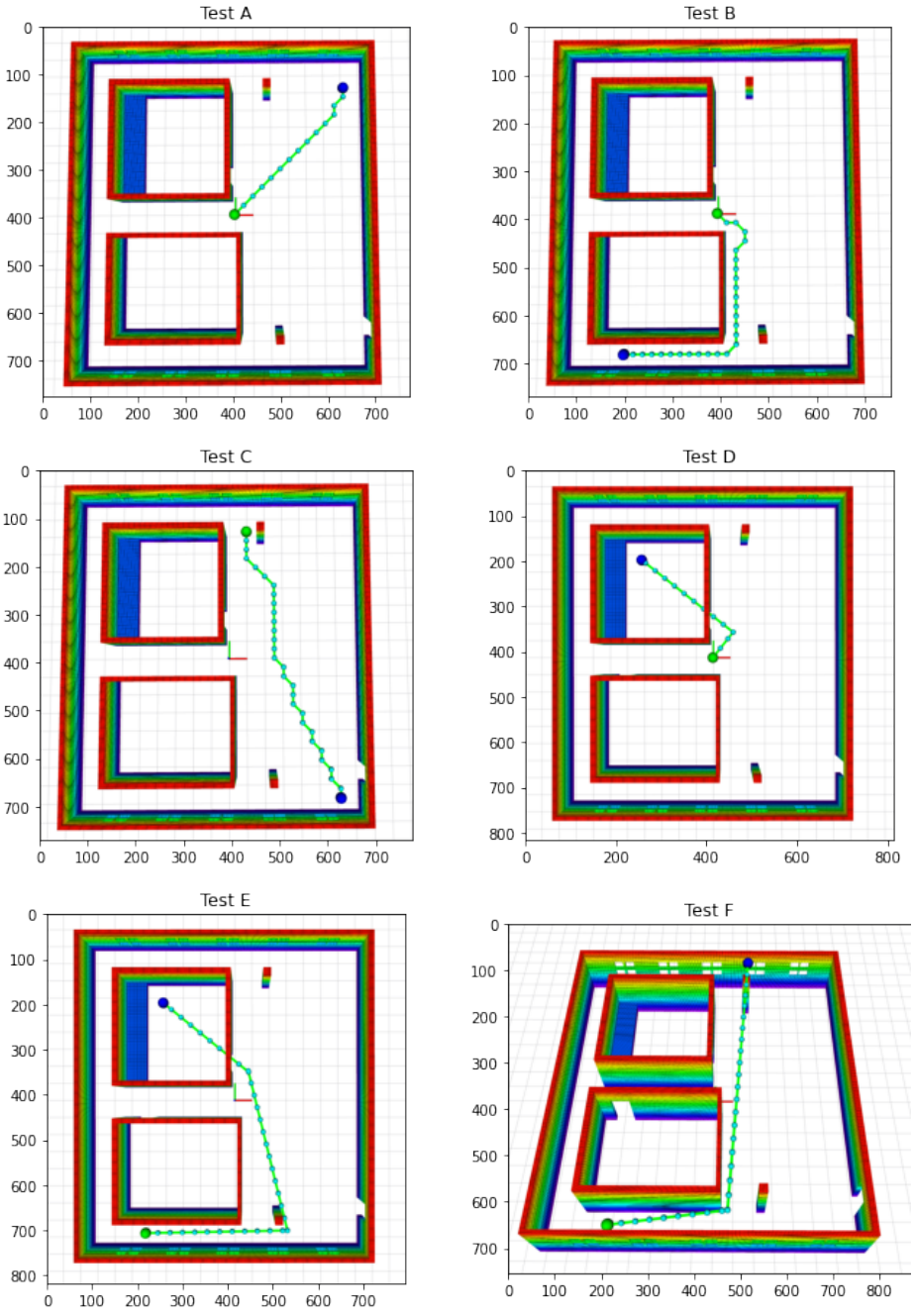

Figure 1: All scenarios on which the three algorithms were tested

Since A* is a deterministic algorithm, we only need to run it once for each scenario. With RRT*, we get a different result every time we run it. Therefore, we run RRT* and RRT* optimized 10 times each. We set $i_{max} = 100\,000$ iterations and $i_{min} = 500$ iterations. To evaluate the performances of the three algorithms, we measured five different parameters:

- **Time performance**: Measures how long does the full path planning algorithm to complete. On their own, the values are not useful since the UAV hardware will be different, but they can still be relatively compared to each other.
- **Path curvature**: A robot usually needs a smooth path to fly easily along it. Smoothing algorithms can be applied to soften the path curvature, but an already smooth curve can make things easier.
- **Path length**: The path produced needs to be as optimal as possible. We defined the length as the sum of the distance between each waypoints of a path.
- **Distance between waypoints**: In order to use the PX4 Offboard mode, we need waypoints close to each other.
- **Octomap checks**: During the path planning, the algorithm may want to check if a point is occupied or not. To do so, it calls the octomap API, which may be a computationally expensive computation, which we want to monitor.

6.2 Time Performance Comparison

In this section, we compare the processing time of each algorithm. The processing time corresponds to the duration between the beginning of the call to the path planning algorithm and the return of the computed path to the rest of the autopilot logic. It also includes the path post-processing.

First, we compare the global processing time for all scenarios, for each algorithm. We obtain the result in figure 2. We can see that A* has a higher mean than RRT* and RRT* optimized, but its standard deviation is much smaller. It looks more consistent than the other two, which have a lot of outliers. RRT* optimized is also a bit longer on every test cases than RRT*, as one could expect. On individual tests though, this effect is not visible, which can mean that the number of tests run of the algorithm is too low. In general, A* appears to always perform better than the worst RRT*.

To extract more information, we can look at the results for each test case. We can see those results in figure 3. In the test A, we can see that A* is better in straight short path. On the contrary, thanks to its random sampling properties, RRT* seems to be better to explore a large portion of the map quickly. For example, in the test C, to avoid small obstacles, RRT* can quickly build an efficient graph. It instantly samples a point which allows it to avoid the small

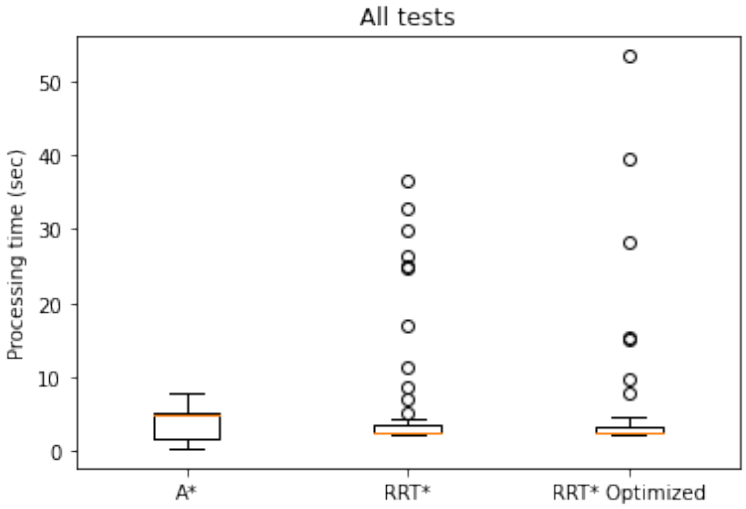

Figure 2: Comparison of the time processing for each algorithm

obstacle. In the same situation, A* has to spend more time to compute a path to avoid the obstacle. However, we can see that in some iterations of the test C, RRT* can be just as slow as A*.

In test D and test E, we introduce narrow passages. There, RRT* performs poorly in some cases, because the nature of the algorithm makes it hard for it to go in a straight line, without getting stuck in an obstacle. However, with the goal in line of sight, like in B and F, RRT* is more efficient than A*.

To conclude, A* may perform less well in some cases, especially when a line of sight with the goal is available. However, it is a lot more stable than RRT*, which can randomly be a lot more than twice the processing time of A*.

6.3 Path Curvature Comparison

In order to be usable by a robot, we need some constraints on the path. Especially, it should not have too sharp angles to be able to turn while still moving forward. Most path planning strategies include a smoothing algorithm. We did not incorporate one. So monitoring the variation of the angle along the path computed may not be relevant for the final robotic application. However, it can still give us an idea of the type of output produced by each algorithm. To evaluate this metric, we measure the relative angle between the two directions given by the waypoints of a path.

We can see the results of our work on all situations in figure 4. Because A* is applied on a grid graph, it is normal that it stays quite consistent, with a lot of straight paths and some few turns. However, this property of the A* algorithm makes it do a very sharp turns (either 45 or 90), which can be hard to use. Especially in straight but diagonal path, A* has to do small turns which are not

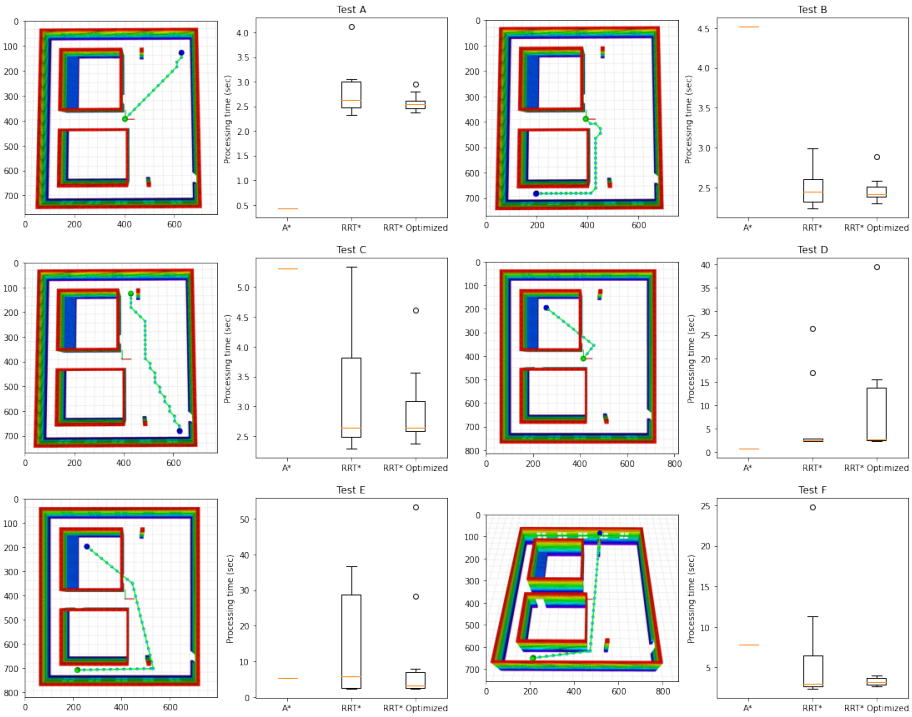

Figure 3: Comparison of the time processing for each algorithm in each test cases. For each column, we show the test case on the left, with the start in green and the goal in blue. The path drawn is only for visual purposes.

efficient at all. We could imagine some post processing to apply to the path to improve it. Additionally, a different and smarter graph generation algorithm, like a Voronoi graph, could soften the curvature of the path.

On the contrary, RRT* without any optimization is doing, as expected, a lot of turns in every directions. This is a result of the randomness of the algorithm. However, re-routing the tree is probably helping to keep a coherent structure. This is why we still see a mean angle of rotation relatively low. We could expect the classic RRT algorithm to work a lot worse compared to RRT*.

Lastly, RRT* optimized is mostly made of straight paths thanks to the over-sampling function applied during the post-processing step. Thus, more than the majority of angles are 0. However, it will be made of a few random turns. Therefore, this algorithm seems to be quite equivalent to RRT* without optimization, with strong angles going in every directions.

To conclude, for each algorithm, we will need a smoothing algorithm to greatly improve the quality of the path.

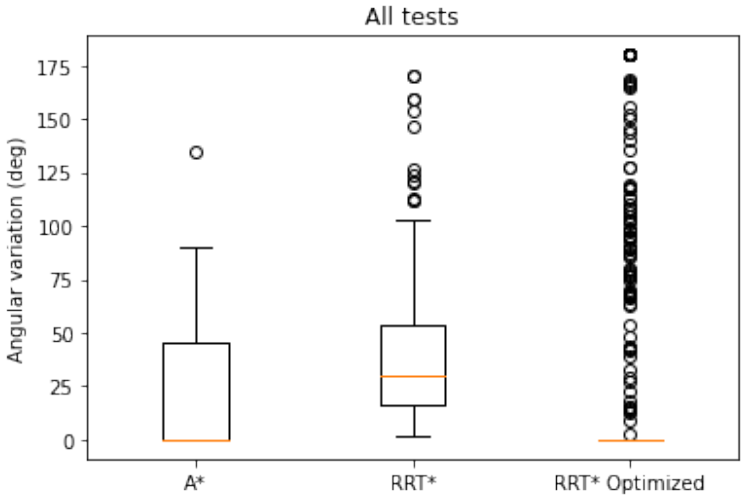

Figure 4: Comparison of the angle along the path for each algorithm

6.4 Path Length Comparison

In this section, we compare the length of the final computed path, where the length represents the sum of the distance between each waypoint of the path. Additionally, to normalize each length with respect to its test case, each length is relative to a straight line path from the start to the goal points. The formula for the relative length is $\frac{path\_length}{straight\_line\_length} - 1$.

We can observe the result of the comparison in figure 5. As one could expect, A* is relatively better in a lot of cases. The means of the algorithms are very close, but RRT* has a large standard deviation and a lot of outliers, because of its sub-optimality in finite running time. Lastly, RRT* optimized is slightly better than RRT*. We can see here how the small optimizations done to the path can greatly improve the results of the algorithm, without increasing too much the processing time.

We display the results for each test cases in figure 6. As shown in tests A and C, the main weakness of A* is in diagonal straight paths. On the contrary, RRT* immediately connects to the goal point if it is in line of sight with a node of the tree. We can even see in those examples that RRT* optimized reaches almost always the optimal path, which is a straight line for the test A, and a near straight line for test C.

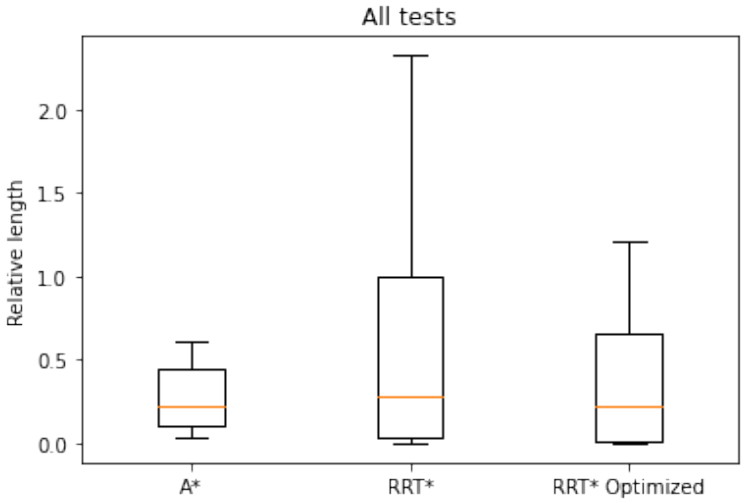

Figure 5: Comparison of the relative length of the path for each algorithm

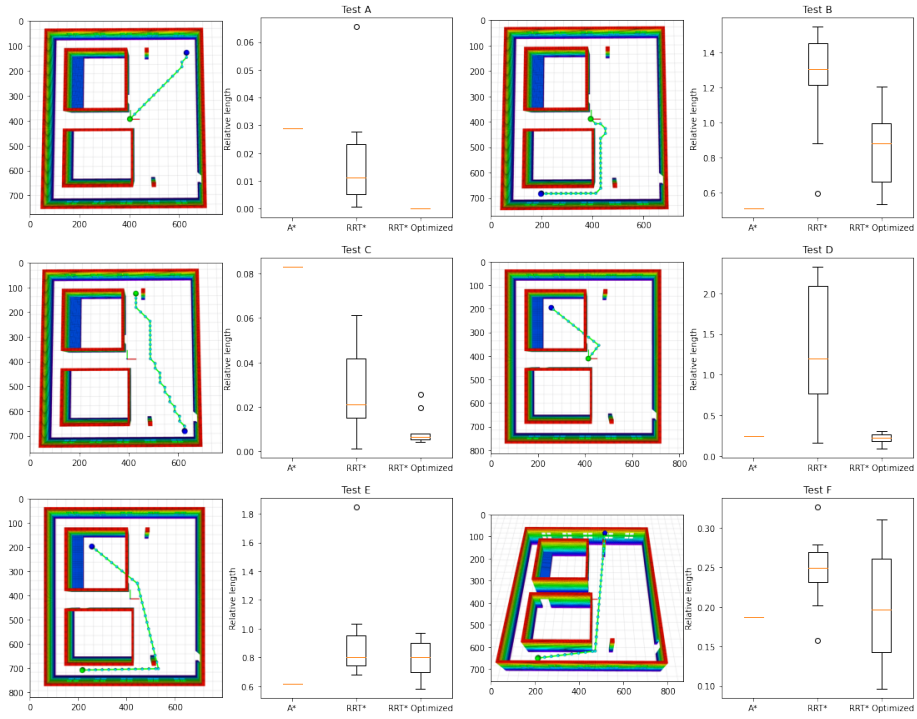

Figure 6: Comparison of the relative length of the path for each algorithm in each test case. For each column, we show the test case on the left, with the start in green and the goal in blue. The path drawn is only for visual purposes.

In every tests, RRT* without optimization can produce a lot of inefficient paths. In tests B and D, we can see how bad the answer of the algorithm is. As expected, RRT* has difficulties in navigating close to walls. However, we can see that the added post processing can greatly improve the length and remove unnecessary turns that the RRT* is computing. In test D, we can almost completely compensate the poor RRT* result and reach the same level of optimality as A*.

In cases with narrow passages, like doors, in tests B, D and E, A* is almost always outperforming RRT*. Indeed, going in a straight direction is not an easy task with a sampling based planner. But the path optimizations can still reduce the gap between the results.

Lastly, in case F, in 3D, RRT* optimized is sometimes outperforming A*, which may be because of the straight line situation, which may be even worse for A* in 3D.

6.5 Distance Between Waypoints Comparison

In this section, we compare the individual distance between each waypoint of a path. We can see the results in figure 7.

A* and the RRT* optimized algorithms are outperforming by a huge margin the simple RRT*. This is expected since the optimized RRT* especially introduce an over-sampling function to reduce the space between each waypoint. A* is also extremely efficient, because it discretize the space in grid a finite size. So by reducing this parameter, we can reduce the waypoint distance even more, at the cost of a big increase in processing time.

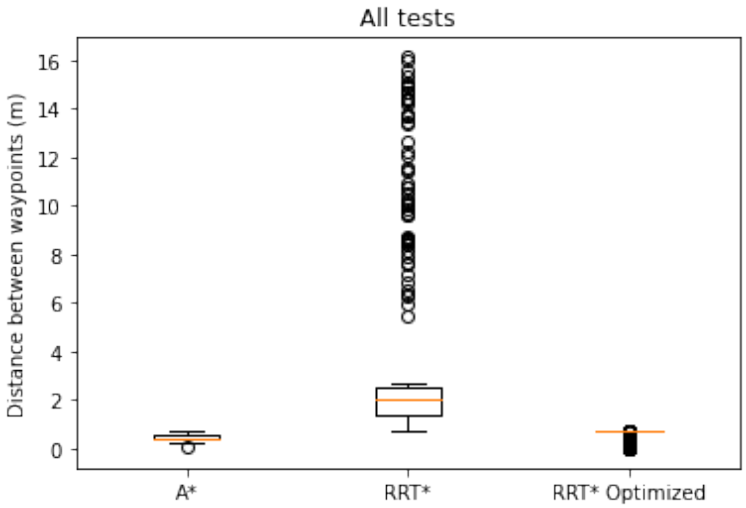

Figure 7: Comparison of the distance between each waypoints for all path for each algorithm

6.6 Octomap Check Comparison

During the execution of the path planning algorithm, it sometimes needs to check whether a point is occupied by an obstacle or not. This is done through the octomap API, retrieved from the ROS Octomap node, as explained earlier. However, this sort of function call can be expensive, especially if the environment is large or finely detailed. Therefore, we monitor the number of calls done to the octomap API, which should ideally be minimized. We are especially interested to see how the number of octomap checks may affect the path planning algorithm.

We can see the result of this experiment in figure 8. Instantly, we see that A* is always the most efficient algorithm in terms of octomap API calls, with a very small variance. On the contrary, RRT* is extremely less efficient and can make up to 400 000 function calls. In a more complex environment, it might cause some performance issues. We can explain this behavior by the fact that, for every re-routing, a lot of calls to the octomap API have to be done. This increases exponentially as the number of neighbor nodes increases rapidly.

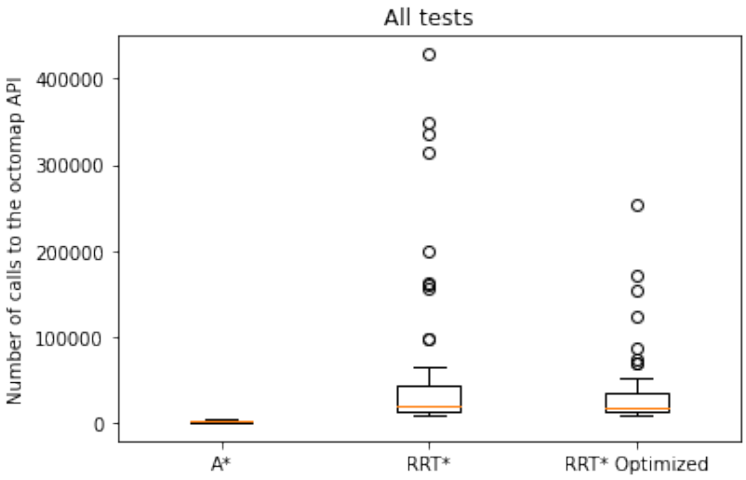

Figure 8: Comparison of the number of calls to the octomap API for each algorithm

Surprisingly, we would expect the optimized RRT* to make slightly more calls than the simpler RRT*, since, to execute some post processing optimization, it needs to execute a lot of collision checks. Therefore, the number of tests done is probably not enough to correctly evaluate the relationship between the two algorithms, as we have seen previously in various results.

Additionally, when looking at the number of calls for each test cases, we can see that there does not seem to be a correlation between the number of API calls and the time performances graph. Thus, we can safely assume that, in this setup, the calls to the octomap API are not too computationally expensive,

contrarily to our expectations.

7 Conclusion

In this research work, we have seen how to implement a complex robotic navigation pipeline. With a UAV, we are using the PX4 firmware along with the ROS middle-ware, which handle communications between all robotic components like the simulator Gazebo, the visualization tool Rviz, the MAVROS node, the octomap server and our software. Using this simulation infrastructure, we built a pipeline to obtain a ground truth octomap of the simulated environment. Finally, we implemented a ROS node to compute a valid path between a start and a goal point, and fly the UAV safely to the goal, using the Offboard mode of the firmware.

To decide on which path planning algorithm to implement, we setup few experiment to measure their performances. When comparing A* and RRT*, we found some similarities and advantages to one algorithm or the other depending on the situation. On the one hand, A* is more stable and will output an almost optimal path in most situations. On the other hand, RRT* evolves in random direction, which makes it ideal in large, non-cluttered environment. In narrow passages, the random sampling of the algorithm makes it hard to efficiently use. For an optimal use case, RRT* needs a lot of processing, but can then be very efficient. In every cases, both algorithms need path post-processing to be used in a real-world application, in order to remove unnecessary turns and smooth the path curve.

While A* is more robust and adaptable to any situation, especially in a dynamic environment, RRT* may be a lot more useful when the goal position is unknown. If the starting position is always the same, one could also pre-process the environment using a RRT* algorithm with a lot of iterations. Then, with just the goal position, determining an optimal path can be done easily with a time complexity of $O(n)$, which makes it more efficient than A*.

For future work, we will continue to improve the UAV autopilot, especially by using sensors data to progressively build the 3D map from incomplete information, instead of using an octomap ground truth.

The source code for this entire project is available on Github: `https://github.com/rhidra/autopilot`.