ME 46060 FINAL PROJECT: OPTIMAL PATH FINDING FOR CUBESATS ON FRICTIONLESS TABLES

Pol Francesch Huc

07/05/2020

Contents

INTRODUCTION

Background

The purpose of this project is to generate an optimal path finding algorithm for my research endeavors at my home university. There, I was part of a team building CubeSats to do obstacle avoidance maneuvers in a friction-less table.

A CubeSat is a small satellite that is composed of different units - or U's - which each measure 10x10x10 cm. A 3U CubeSat for example, would measure 30x10x10 cm and is practically composed of 3 1U CubeSats stacked. These small satellites allow for cheap ways to test technologies in space. In our case, we are working with a 3U Cubesat, however this is easily modifiable in the software.

In order to test satellites before launching them, many companies and government agencies have friction-less surfaces. These are made of polished granite that is chemically treated, and produces a negligible amount of friction, in most cases. In the case of CubeSats however, as they are rather small and have propulsion systems that produce very little thrust, this friction is no longer so negligible, and usually has to be accounted for in modelling.

Our CubeSat is equipped with four cold gas thrusters to be able to move in four directions, and a reaction wheel which allows it to spin. It is sitting on top of a levitating surface, which blows air down onto the friction-less table to generate lift. This surface is roughly circular, and is somewhat larger than the CubeSat.

For communications with the ground station, our CubeSat uses laser technology, which requires line of sight to operate. If an obstacle is to come between the CubeSat and the ground station, the CubeSat will lose the communication link. This is not much of an issue, as it wil continue past the obstacle and then re-synchronize with the ground station.

Generating Scenarios

In order to make sure the optimization algorithm is versatile, we need to be able to create obstacle maps algorithmically. For these, we can specify the size of the map, the obstacles, and the resolution of the map. The resolution is an interesting property we will be analyzing later on. As a pre-cursor, the optimization algorithm chosen breaks the map into nodes, the resolution parameter specifies how many nodes (or cells, if we wish to talk about matrices) per meter we will have.

I must note however that certain restrictions must be placed in order to minimize the amount of times that a path may not be possible. I came up with the following example to illustrate some limitations. The table is four meters wide, the three obstacles 1 meter wide, and the CubeSat + levitating surface are 1 meter wide. If the three obstacles are in line, they will take up 3 of the 4 available meters. This makes it theoretically possible to pass the CubeSat, but practically this is not a useful solution and no path is possible (this is as we prefer to have a safety margin around the CubeSat in order to account for perturbations in the movement). Hence, when creating these maps, it is easiest to visually check them to make sure there is a possible path.

It is true that you could theoretically create maps randomly, however it would be difficult to check, using an algorithm, whether they can be solved without actually trying to find a path. Here then, we would run into the issue of not knowing if it is the map that is unusable, or if the algorithm we have created is not good. To avoid this sort of issue, I generated some maps myself and tested the algorithm with that. Below is an example of one such map. Note that the axis seem to be rather large, however this is not the case. This map is showing the grid of the matrix, which may have more than one cell per meter.

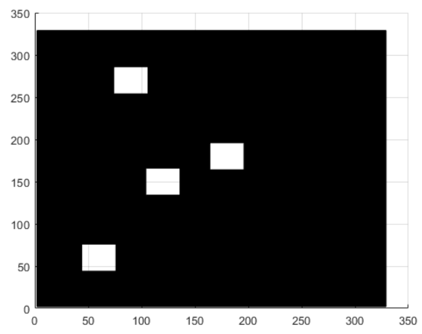

Figure 1: Map with 4 obstacles

When plotting the final results, however, I tend to use another function, which switches the color palate and allows us to view the obstacles better. It also displays the start and goal, and has the axis in meters. This is not used to represent the map, as it does not work well with the optimization algorithms, and is more of a manipulation of the plotting tool on MATLAB.

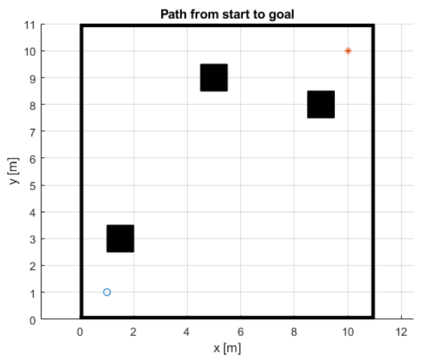

Figure 2: Map with 3 obstacles, and start and goal

PROBLEM FORMULATION

In this section I will be properly describing the optimization at hand, and giving it a mathematical definition. Additionally, I take some time to discuss certain modelling restrictions which are imposed on the problem in order to reduce complexity and improve the speed of the solution.

Optimization Problem

In short, the goal of this project is to find the shortest distance between two points when there are obstacles which must be avoided.

$$\begin{aligned} \min_{\sigma} \quad & J(\sigma) \\ s.t. \quad & \sigma(0) = x_{init} \\ & \sigma(1) = x_{goal} \\ & \sigma(\alpha) = x_{free} \qquad \forall \alpha \in [0,1] \end{aligned} \tag{1}$$

$\sigma(\alpha)$	=	continuous function [0,1] which spans the whole area
$J(\alpha)$	=	cost function (e.g. path length)
x_{init}	=	initial CubeSat configuration
x_{goal}	=	goal CubeSat configuration
x_{free}	=	the free space in the area (not occupied by a wall)

This is a rather general optimization problem. We will further explore this definition when I show my own optimization algorithm.

Modelling Aspects

In order to properly model this problem, we would be required to experimentally measure certain elements such as the forces generated by the CubeSat - thrust and rotation -, some physical quantities - e.g. mass, moment of inertia - and the friction. Due to the restrictions from COVID-19 this is not possible, and hence we must either assume quantities or reduce the complexity. I opt for the second option, as it will still give a reasonable result and will generate a proper path.

In this case then, we are only concerned with the distance covered, not the amount of time it takes to cover it. This may be problematic, as the speed of the CubeSat is not constant and it is not possible to integrate to get the time. One clear issue with this method is that our CubeSat prefers to stop and then turn, hence adding a lot of overhead time to any turn. This is not taken into account by path planning.

I must also note that the way the algorithms handle obstacle and CubeSat sizes is not very friendly. Hence, I simply made the following determination:

obstacle sized used by MATLAB = obstacle size + CubeSat size + safety margin

This makes it much easier to handle the different sizes without having to alter the map unnecessarily. In the code, this will always just appear as the obstacle size, and the CubeSat will be treated as having size of 0.

We may also be interested in inserting certain path requirements. For my specific research project, the goal was to maintain laser communications with a ground station as much as possible. This is not a large issue, but it is preferably avoided. Hence, we could add a penalty to having line of sight broken in our optimization. This will not be treated in the report, but is a possible avenue for further research.

To summarize, we will treat the CubeSat as a black box which can move in any direction as needed, and has no size. We will instead incorporate the size of the CubeSat into the obstacles such that less work is needed to compute this. I ask that you keep this in mind when analyzing the results of the optimization. It may look as if the CubeSat is passing too close to an obstacle, but in truth, this has already been taken into account.

Optimization with MATLAB Path Planning

In this initial optimization we use the MATLAB path planning tool to investigate their results.

Motivation of optimization approach and choices

This allows us to get a good sense of how path planning is handled in MATLAB and I can begin to generate my own path planning tool.

I will be using the Probabilistic Roadmap (PRM) path planning. This algorithm places random nodes on the map and then tries to link all the nodes to each other. If a link between the goal and a node is possible, it will then search search the shortest path between the nodes. The limitation here, is that the nodes are placed randomly, and so a global optimum is not necessarily obtained.

There is one issue with these functions however, and that is that one of our design variables - the number of nodes - is a discrete value. For this reason, we cannot apply standard optimization algorithms. For this reason, we will only be using it as an exploration of MATLAB's native capabilities when it comes to path planning.

Obtaining the path

This step is rather simple, thankfully. MATLAB does most of the heavy lifting here, and the algorithm is already written. All the user can specify is the map, the number of nodes, and the connection distance - distance between the nodes. See the code in the corresponding zip file for this section for more information.

Note, for this section I am using some pre-loaded MATLAB maps, I do this as they are readily available and have been especially created to work with these functions. Hence, it is easy to use them. The results are still useful as we are looking to describe the behavior of the algorithm qualitatively, and compare how it acts to my self-made one.

Here are some results at varying resolutions:

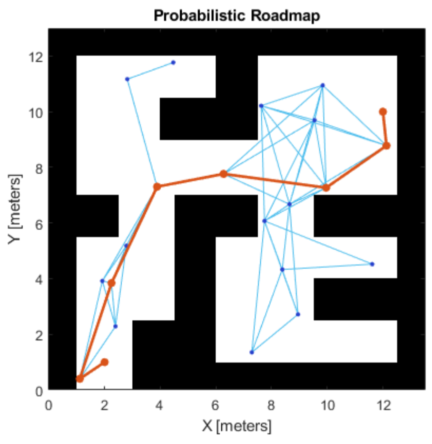

Figure 3: PRM with 20 nodes

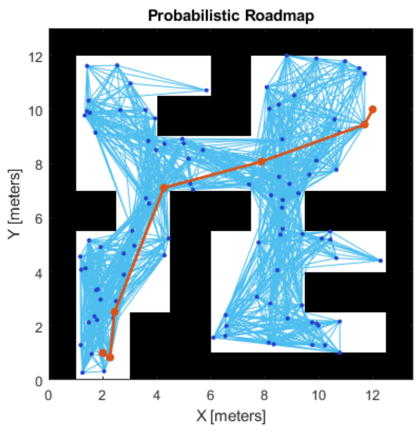

Figure 4: PRM with 100 nodes

Here are results at repeated resolutions to display effects of randomness:

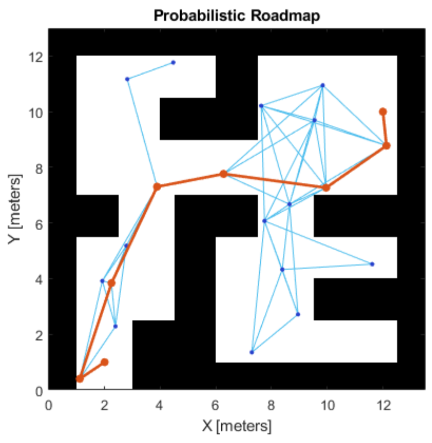

Figure 5: PRM with 20 nodes

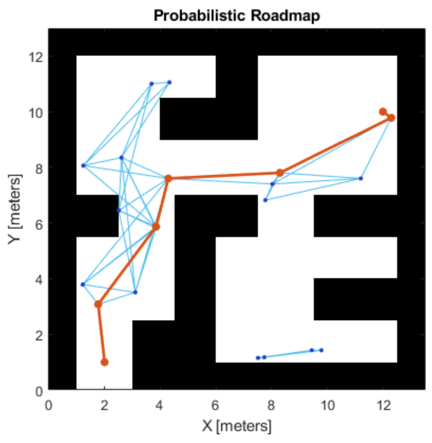

Figure 6: PRM with 20 nodes; different attempt

Observations

The obtained optimum using PRM has certain issues. Firstly, it is based on the random location of nodes, hence, a consistent solution is not generally possible. Second, the cost of running the algorithm dramatically increases as we add more nodes, as MATLAB will check every possible connection before making the shortest path.

I also note that this solution will make you take weird directions to get from point A to B, which is due to the random node locations of course. This is especially true for the lower node counts, but even for Figure 4 we can see some of this behaviour at the beginning.

Conclusions

The solution given by the PRM algorithm from MATLAB is usable, and will certainly get you from point A to B. However, we are looking for a better solution than this, which does not depend on random node locations. Hence, we must move onto a more advanced algorithm which will hopefully suit our needs better.

Optimization using A* Algorithm

Here, we seek to get a better path than before using an A* algorithm for path planning. I have implemented the code to run this algorithm myself, and it results in some rather interesting paths.

Motivation of optimization approach

We use an A* algorithm here as it is relatively easy to implement, and still gives good results. Unfortunately, implementing good path finding algorithms is rather time consuming, and would require much longer to do. The A* algorithm is hence a happy medium between the PRM and the more complex alternatives.

Explanation of A* Algorithm

The A* algorithm takes a map with obstacles and finds a path between the start and the goal. It does this by first, breaking up the map into nodes or cells, and represents it as a matrix. It then begins at the start and searches its immediate neighbors for what is the perceived cheapest next node.It then moves to the cheapest node. This last bit is important, as the algorithm can spread and have different branches. Hence, if it is working on one branch, and that becomes more costly than another, it will automatically switch.

If the algorithm finds the goal through this method, it will then work backwards to get a path. In some cases, the code does not store a node's parent (so where it came from), but in my case it does. This makes the walking back portion rather straight forward.

With this knowledge in hand, we can now redo our optimization problem formulation as follows.

$$\begin{aligned} \min_{n} \quad & f(n) = h(n) + g(n) \\ s.t. \quad & p(n_0) = x_{init} \\ & p(n_1) = x_{goal} \\ & p(n_\alpha) = x_{free} \qquad \forall \alpha \in [n] \end{aligned} \tag{2}$$

n	=	discrete matrix representation of the area
α	=	any node in n
$h(n)$	=	heuristic cost from n_α to goal
$g(n)$	=	cost to reach n from start
$f(n)$	=	estimated cost from start to goal if path goes through n
$p(n)$	=	path taken by algorithm
x_{init}	=	initial CubeSat position
x_{goal}	=	goal CubeSat position
x_{free}	=	the free space in the area (not occupied by an obstacle)

Plotting Methods

I must also note that I developed two plotting methods, that I will demonstrate as follows. This is reminiscent of the Generating Scenarios section from earlier in the report, however here I am showing the path, not just the map. One, which displays the whole tree as it works, and the other which simply plots the final path. The former can be very useful to identify certain behaviours and confirm that the algorithm is working. The downside is that it is very costly to run, and can take minutes at a time as we must plot new points for every new node that is interpreted. On the other hand, the second method works much faster (usually under one minute for reasonable resolutions) for the same map. This can be very useful for doing error analysis, and getting a better view of the optimal path.

The "free space" in this one is in black, and the obstacles are in white. The opposite is true for the other plot (which only shows the path) as I find it to be more visually appealing.

Additionally, I have included some videos of the algorithm working it's way through the more complex map so that you can view this process without having to open MATLAB.

Sample solutions

First let us look at a simple example. I will show both of the plots here so their differences are clear. The first will show all considered nodes in green, and then it will mark in red the best path out of those nodes in red. The second will only show the path, and will have the switched color palate.

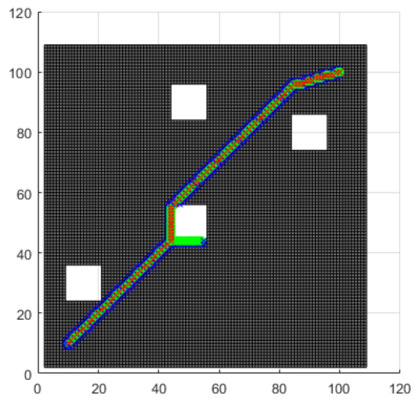

Figure 7: Solution showing detail about optimization

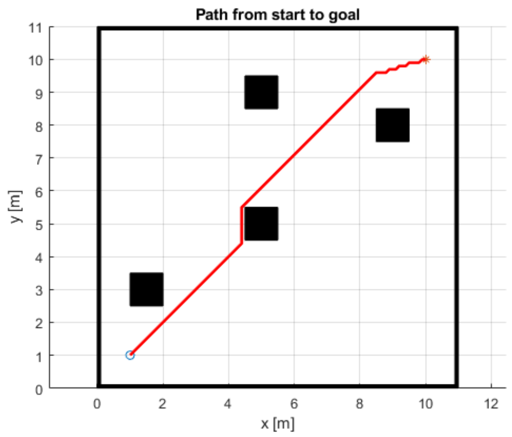

Figure 8: Solution showing only final path

Next we will look at a more complex path. I will still show both of the plots as the first is much more informative on behaviour, but it is difficult to read.

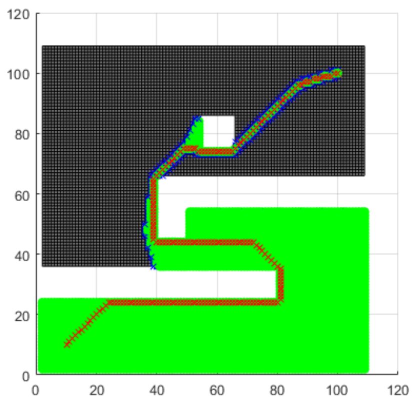

Figure 9: Solution showing detail about optimization

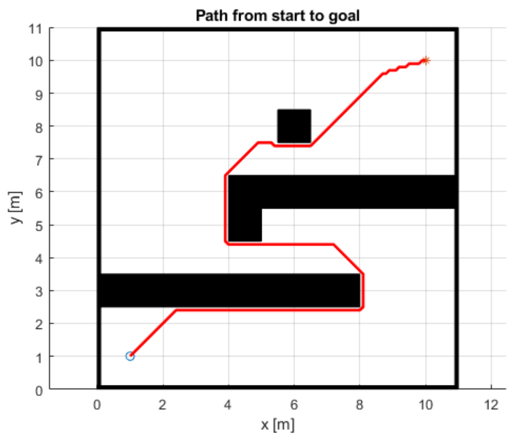

Figure 10: Solution showing only final path

Observations

From this we can immediately tell that the A* algorithm is also not giving the optimal solution. This is as it only considers local information when deciding the cost of a node, and does not consider global information.

This is clearly due to the configuration of the heuristic cost - h(n) - which is simply calculated as the Cartesian distance between the node and the goal. It does not take into account, for example, how the path may have to avoid the obstacle, and hence doing so earlier, would actually result in a shorter path. We can somewhat adjust for this using a weight which determines how greedy the algorithm is. Such that

f(n) = h(n)+g(n) becomes f(n) = weight*h(n)+g(n).

Note that, if we increase the weight, the algorithm will become greedier, and hence the solution will not be as optimal. That being said, if we decrease the weight, we also increase our computational cost, as the algorithm will be much more willing to explore around it, and will not focus as much on trying to reach the goal.

Finally, A* algorithms are best-first algorithms, meaning that they assume that the first solution they find is the optimal one. Clearly, this is not the case, but it does results in solutions which are good enough, and reasonably better than the PRM method.

Investigation of weights

Here we will show how varying the weights can have an effect on the solution. It is most interesting to look at the plot which shows all of the nodes which were explored, as that will show us how the algorithm arrived at a specific path.

We will repeat the same optimization process as Figures 9 and 10, so we will only show the results of compared to those figures. In that case $w = \sqrt{2}$.

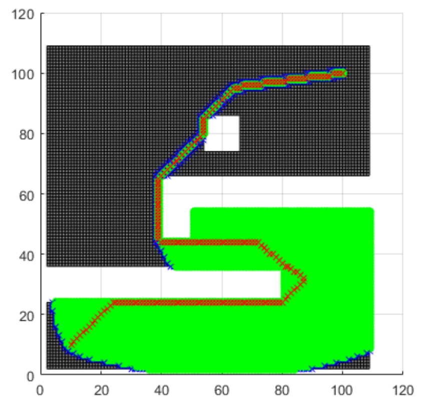

Figure 11: For w = 3

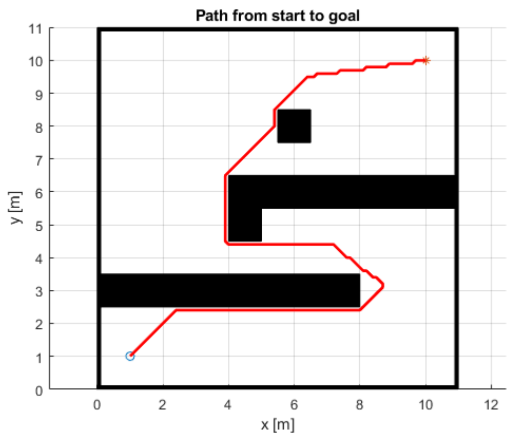

Figure 12: For w = 3

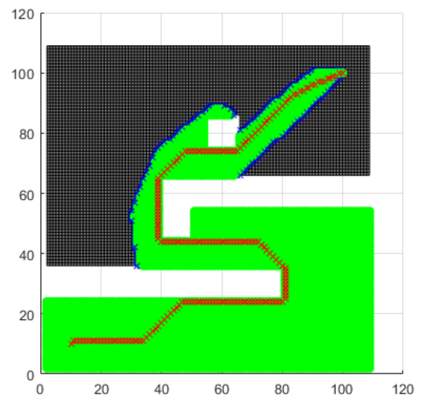

Figure 13: For w = 1

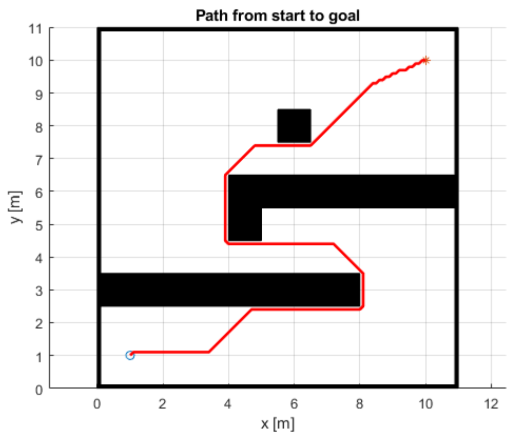

Figure 14: For w = 1

Interpretation of results

We can note certain clear behaviours from these plots (please note that these images are only final results, when running the code, it will show how the algorithm progresses through all the points making the process slow as it must re-plot everything every time it considers a new point). First, it is clear that the algorithm is quite greedy by design. Even at not so high weights, it still places a lot of importance on going to the goal in a straight line. This is as the algorithm is only given local information, and is unaware of walls until it hits them. This makes it a bit inefficient, but unfortunately, is one of the limitations of the A* algorithm.

As we become greedier, we can clearly see that the algorithm does not seek out nearly as many possibilities. And does reach the goal faster - in terms of computational time. The final solution does change a bit, but I would argue this is not very significant between w = 3 and w = $\sqrt{2}$. Intuitively, the difference in the distances is minimal, and this is backed by the code. Evidently, if we had an infinite amount of time and we wanted the best distance this algorithm can come up, then we would lower the weight. Unfortunately, the cost of running the program is important, and so it must be taken into account.

That being said, we can ascertain that the non-greedy algorithm definitely approaches a better solution. We can clearly see this in the bottom portion where it passes much closer to the obstacle.

Conclusions

The A* algorithm certainly gives a better solution than the PRM algorithm with which we began, but it is still not perfect. When we look at these results, we can intuitively say that that is not the optimal distance as it takes hard turns and does not have much future perception. Nevertheless, we can indeed improve this algorithm somewhat by properly choosing a resolution to work with, which is something we will explore in the next section.

Efficiency Analysis

In this section, we will visualize the added cost of increasing the resolution of our model.

Method

Here, I ran the optimization code on the same map shown below for different resolutions. This allowed me to estimate what the best resolution to be working in is, as it will give me an idea of the relationship between computational time and distance covered.

I also did this for a couple of weights. I did not complete this efficiency analysis for very low weights, as we know that their computational cost is rather high. I am not really interested in this as I can roughly determine how the different weights will act by having testing three, and extrapolating from there.

I am limiting this, as the total computational time is very long. I am not only running it for about fifty resolutions (only running for even resolutions), but I am doing so fifty times so that we can determine an average for each resolution. This adds lots of time to our total run time.

Sample Solutions

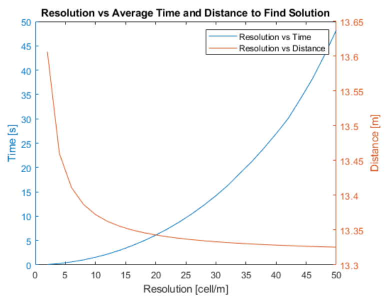

Figure 15: For w = 1

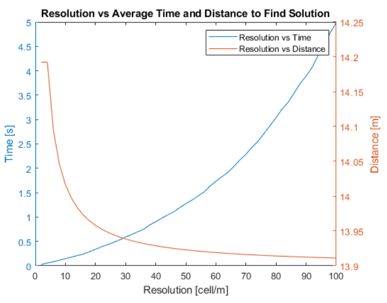

Figure 16: For w = $\sqrt{2}$

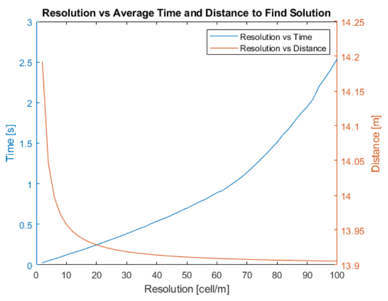

Figure 17: For w = 2

Conclusion

From the plots we can see that a resolution between 20 and 30 is good for all of these weights, though it is important to note certain differences. First, though it's not very clear, the total distance covered is lower for w = 1 at lower resolutions than for the other two weights at high resolutions. This shows us the power of the weight. Which can significantly improve our solution. That being said, the difference in distance is not very large compared to the difference in computational time. For the most part, the larger weights did not go above a time of 5 seconds, however at w=1, our larger resolutions take 50 seconds each to run.

It is for this reason, that the w=1 plot only spans until resolution 50, as running any more is much to costly and could take days due to the exponential growth.

Conclusion

We can see from the results that the A* and PRM algorithms are both possible choices when attempting to determine the path to take on a given map. It is clear that the A* algorithm will give better results, though the increased cost has not been explored.

The A* algorithm has certain drawbacks that are there by design. Firstly, the heuristic distance is a tricky thing to calculate, so I left it as simple as possible and made it the distance between the node and the goal. If we ignore the heuristic distance, we arrive at Dijkstra's algorithm, which has a much increased computational cost, as it has no preferred direction.

The second clear draw-back of the A* algorithm is it's best-first philosophy. This, admittedly, is great for computational time as it simply takes the first solution and returns it as the optimal. Clearly though, it does not always return the global optimum.

Recommendations

Some follow up research could involve the improvement of this algorithm to better fit the CubeSat project, or some work in implementing a new algorithm that makes decisions based on global information rather than local.

For the first of these, I mentioned at the beginning of this paper that the CubeSat must try to maintain communication link - for which it requires line of sight - with its ground station. This could be achieved by modifying the heuristic - which estimates the cost from a node to a goal - such that it is more costly to go on one side of an obstacle than another. This would be rather difficult though, as it would have to be scaled such that the heuristic is still relevant but not overbearing (ie not too small or too big) relative to the total cost. The simplest approach for this would be to modify the heuristic such that one side of your free space is cheaper than the other (the one closer to the ground station of course). Such that if faced with an obstacle straight one, it will choose to go on that side. It is difficult to know how well this would work, but it is certainly a possibility.

Furthermore, I stated that it is rather expensive to turn with the CubeSat. This is as it is a rather unstable system, so we would prefer to stop and then turn. Hence, though the total distance may be shorter if we have lost of little turns, or turns which require moving, this would be more expensive in real life as we have to stop. One simple solution would be to apply a filter such that only the larger movements of the path are kept and the smaller things are ignored and replaced with a straight line. We could further complicate this by changing our objective function to add a realistic cost to turning.

Finally, we could also move on from the A* algorithm into something else, which may provide a better path. It is my understanding that applying a genetic algorithm, such as reinforcement learning, is feasible in this case, and there are previous implementations. This would provide a better solution, as once could more feasibly integrate the above mentioned drawbacks, but it would greatly increase the implementation time of the algorithm.

BIBLIOGRAPHY

Theory Sources

Patel, Amit, "Amit's A* Pages." Stanford Theory Group, 09 May 2020. URL: http://theory.stanford.edu/ amitp/GameProgramming/

Code Sources

Baijayanta, Roy, "A-Star (A*) Search Algorithm." Medium, towardsdatascience.com, 2020. URL: https://towardsdatascience.com/a-star-a-search-algorithm-eb495fb156bb

Chrabieh, Anthony,"A star search algorithm." MATLAB Central File Exchange, 2020. URL: https://www.mathworks.com/matlabcentral/fileexchange/64978-a-star-search-algorithm

MATLAB, "Path Planning in Environemnts of Different Complexity." MathWorks, 2020.

Premakumar, Paul, "A* search for path planning tutorial." MATLAB Central File Exchange, 2020. URL: https://www.mathworks.com/matlabcentral/fileexchange/26248-a-a-star-search-for-path-planning-tutorial

Appendix: Read Me for Code

Due to the many functions I am using in this code, I have split the .zip file which goes along with this document into different folders, each pertaining to a section of the document. Here, I will describe each section and provide instructions on how to run the code. Note that the section names in this appendix roughly match the section names in the paper, and in the zip file. Along with the code, I will include the image files which are embedded in this document. Please note, it is likely that you will have to install additional optimization libraries if you have not used these functions in the past.

Making Maps

I am including this section for completeness, so that you can see the different algorithms. The one which returns the ugly maps is "createMap2". The one which generates the nice map is "plotOptPath", which is mainly used for plotting the path in a nice map. You can feel free to generate your own maps using createMap2.

MATLAB Path Planning

This includes a single script file which you can run, and will return a map with a, hopefully, complete path. When this is turned in, the algorithm will be using 20 nodes, which is just enough to sometimes make a path. If when you run it, it does not make a path, I recommend trying again. For consistency, you can change line 18 to a higher node number.

A* Algorithm

This section represents the bulk of the work of this paper. The main program is "aStar_Optimization", and it will run the optimization code for you. At the top, you can see various objLoc you can choose from; I used these in my testing. Feel free to make your own if you wish.

On line 24, you can select the weight which you would like to use. And then on lines 26-34 are where you select the type of graph you would like. If you want to see it in action, only line 27 should be uncommented. If you are only looking to see the final path, only lines 30, 33 and 34 should be uncommented.

I have included some videos in this section to demonstrate the "Fancy plot"

Efficiency Analysis

The main code here is "checkBestRes2" but producing the plots I have generated for the report can take hours, so I would not recommend actually trying to run this code unless you can leave it be for a while. If you want to check that it works, modify the code to run through a small amount of resolutions, a small amount of times. In this case, we would modify lines 22 and 37. I would also recommend using a reasonably higher weight like 2 or 3.