MOTION PLANNING FOR TETHERED ROBOTS

An Undergraduate Research Scholars Thesis

by

MADELEINE PHILLIPS

Submitted to the Undergraduate Research Scholars program at
Texas A&M University
in partial fulfillment of the requirements for the designation as an

UNDERGRADUATE RESEARCH SCHOLAR

Approved by Research Advisor: Dr. Dylan Shell

May 2020

Major: Computer Science

TABLE OF CONTENTS

ABSTRACT

Motion Planning for Tethered Robots

Madeleine Phillips
Department of Computer Science and Engineering
Texas A&M University

Research Advisor: Dr. Dylan Shell
Department of Computer Science
Texas A&M University

Motion Planning algorithms have been studied in many applications, such as network design and robotic motions. However, in the field of Robotics, in some settings the motion might be limited by a cable, preventing the robot from finishing its task. Existence of a cable poses new restrictions on the robot's motion. For instance, the robot has to avoid entangling the cable with obstacles or damaging the cable, while still trying to accomplish its goal. This research aims to provide hardware implementation for an algorithm introduced by Reza Teshnizi and Dylan Shell [5] that plans motions for a robot in such settings. The original publication has only demonstrated the algorithm in a simplistic simulation environment. An extension to this algorithm can be utilized to plan motions for a pair of robots that are connected to one another via cable of fixed length. By connecting these two robots, this imposes a new restriction on where the robots could travel because the robots have to travel together without entangling the tether. This research for the algorithm for two robots is in progress at the moment. The outcome of this project should verify the simulated results in the paper written by Shell and Teshnizi.

ACKNOWLEDGMENTS

I would like to thank Dr. Shell for his support throughout the course of the past two semesters. I would also like to thank Reza Teshnizi, Sophie Hsu, and Albin Kyle Myscich for the help with making this experiment successful.

Thanks to my friends and family and the department of Computer Science for helping make my college experience a great one. I would finally like to thank my parents for allowing me this opportunity to expand my horizons.

CHAPTER I

INTRODUCTION

Background

Robotics would not be where it is today without the use Motion Planning. Without this key element, robots wouldn't be able to accomplish their task in an effective way. The main objective in motion planning is to find a path to move an object from one point to another without colliding with anything given a precise geometrical description of the surroundings. This is no easy task, even with the technology of today, and is in fact NP-Hard [6].

Even though these algorithms are quite difficult to develop perfectly, there are many that help determine the fastest route, which are programmed into the robot. This task can become tedious if the robotic design restricts the robot's motions. This is especially found with robots that do not have free range of their world or a cable is connected to a robot for a specific reason. If these cables happen to be interlaced or becomes damaged by the surrounding area, then this not only prevents the robot from finishing its job but also waste resources. For example, robots that are used to clean up oil spills within a large body of water are connect to a fishing net to gather the oil [3]. If the net becomes interlaced or becomes damaged by the surrounding area, then the robot cannot accomplish its assigned job. Therefore, motion planning algorithms must accommodate for the restriction the tether brings and must still be able to determine the best route within the robot's world. Another example of this type of robot is a robot that uses the connected cable to grab certain items within the workspace. A majority of robots use outer appendages to grab simple items, but this becomes difficult to build and program as items that need to be gathered become more abundant and complex. If a robot is able to use a cable to grab

these items, then the robot can be a relatively simple with minimal hardware. This robot only needs the cable and the specific program that can prevent the tether from entangling, which is more cost efficient and easier to program [1], [2].

This idea of a motion planning algorithm that accounts for a fixed-point tethered robot comes from the paper *Computing Cell-Based Decompositions Dynamically for Planning Motions of Tethered Robots*. Currently, this algorithm is implemented in a perfect world simulation. Our research aims to implement this algorithm in a real-life model. This algorithm accomplishes motion planning by using an inputted 2-D map of the workspace and applies Computational Geometry to these data points to calculate the fastest and safest route for the robot, while still accounting for the tether.

Objective

The purpose of this research is to demonstrate motion planning for tethered robots. We have created a physical system with two robots and an overhead camera to implement the simulation introduced by Teshnizi and Shell in a real-world environment. To accomplish this, we have to handle the complex physical system within the robot, while still accounting for the noise of the real-world.

CHAPTER II

RELATED WORK

Motion planning for tethered robots can be used throughout many industries and the work among researchers spans from moving a singular robot attached to the ceiling to moving multiple ground robots around without entangling. These algorithms combined the use of motion planning and homotopy classes to find the shortest path from A to B, while still regarding the constraint imposed by the tether. In this work, we use the algorithm introduced by Teshnizi and Shell [5]. However, an overview of closely related work is presented here.

A base level algorithm was produced by Hert and Lumelsky's [9]. This algorithm is dedicated to finding the shortest path for multiplied tethered robots without crossing other robot's cables. First, it is determined which robots can become straight-edge robots, or robots that can travel along a straight line between their start and finished points. However, sometimes this does not work for all robots and it must be decided which robots will move straight to their targets and which robots will bend along the other robots. To do this we first go through all the robots and compute their retraction line. This line is calculated by determining the potential path the robot could take around robots that haven't moved. This line is then concatenated with the current configuration cable line and becomes the retraction polygon. If there is a target inside of a robot's retraction polygon, we put a vertex between that target's robot and the original robot into the graph, *G*. The robot with the least number of vertices in *G* goes first and then is taken out *G*. This is repeated until there are no more vertices in *G*. The retraction polygon is then used again to determine the convex hull of each point on the retraction polygon. Each polygon is then travelled along in order along the cable line. This order will eventually become the robot's path.

A robot will go clockwise or counterclockwise depending on if they would go right or left, respectfully.

Igarashi and Stilman's [10] algorithm travels from each neighboring vertex inside of the physical grid and incrementally adds each to the current manifold graph if the chosen position doesn't intersect with an obstacle. Neighboring vertices are defined as vertices that have a Manhattan distance of 1, ensuring that it is the best possible path. A manifold vertex is one that computes the homotopic paths from the cable base to the desired position. This creates a map for the robot of all potential start and goal positions. However, this algorithm needs to be reformed for robots that cannot cross the cable. The way that this is done is by either a) retracting the cable or b) untangling the robot via robot movement, which is done by choosing intermediate targets until the robot is free from the cable bounds. To prevent a possible case of deadlock, the algorithm can also pre-remove manifold vertices that could potentially lead to deadlock, however, this is currently unfinished and waiting for new algorithms.

The closest competing algorithm was produced by Wang and Bhattacharya [11]. The algorithm constructs a visibility graph of all the obstacle's vertices ($v \in V$), the cabled tethered point, robot points (start and end), and all edges between vertices. This graph is then transformed to an *h*-signature Augmented Graph. An *h*-signature is the homotopy class between two points, hence, the original graph converts all vertices to their homotopy class. Their algorithm then uses Dijkstra's algorithm to search these *h*-signatures to find the shortest path for every single path between the cabled tethered points, c_i. A list of all shortest path between tethered points are found in *P* and their homotopy classes are found in *H* which are then used to find the closed boundary of the area. This algorithm can also be amended to account for a workspace that covers multiple task areas.

Grigoriev and Slissenkoy's [12] algorithm first creates an extremity basis over the area. This basis creates a series of cuts along the boundary that will eventually piece together its desired graph, *G*. This graph contains a segment that doesn't intersect the boundary while also touching all the obstacles. This is known as canonical quasi-segments. The algorithm then sets an irreducible word, q_i, to each homotopy class, which is then transformed into a piecewise simple representation, q'_i of the same word. Once every class has a word, then an overarching word, *H*, is assigned to the extremity basis. To determine the shortest path between two points, the algorithm goes through every canonical quasi-segment and determines if it has a neighbor. If this is true, then it concatenates the segments and their respective words. This new word is then transformed into a q'_i and compared to *H*. If q'_i is determined to be a prefix of *H*, then this segment is a part of the shortest path.

Hert–Lumelsky's algorithm is different from the other algorithms because it doesn't employ the use homotopy classes to find the shortest path. However, they do use obstacles as leverage point for the shortest path like the other algorithms. This concept was a good base for the next algorithms and provided an easy way to replace its way of shortest path determination while still using the obstacles to their advantage. This is exactly what Igarashi–Stilman did. They added homotopy classes to determine shortest path but still used the obstacles to get around. Wang–Bhattacharya's algorithm is quite similar, however, was produced after Teshnizi–Shell's algorithm and is best used to see how the current algorithm can be built upon. Finally, Grigoriev–Slissenkoy's algorithm puts another layer on top of the other algorithm's homotopy classes and assigns words to each. This allows them to build upon the classes to determine the shortest path. Each of these algorithms are important to know because they show the state of

current research in motion planning for tethered robots. It also shows that homotopy classes are important concept for motion planning but aren’t necessary for fast algorithms.

CHAPTER III

METHODS

Materials

We want to implement a motion planning algorithm that determines the best route for a tethered robot. To accomplish this, we used two Asus Eee Computers that acts as the primary controller for the robot of the robot and one Desktop PC that acts as a server. Each computer was fitted with the operating system Ubuntu and the software pySerial, which will connect the robot to the Eee computer. To create our 2-D map of the workspace, we used an overhead camera connected to the PC. We used two iRobot creates, which uses the Asus Eee to determine the calculations. Finally, the robots will be connected via a taut cable made of nylon rope.

Theoretical Approach

There are currently three components in this research: the robot programming, the overhead camera processing, and the algorithm. The robot and the overhead camera both create their own 2-D coordinate plane of the world they will be exploring. The algorithm attempts to transform these coordinate planes into one coordinate plane that they both can recognize and adopt. This is represented in Figure 1.

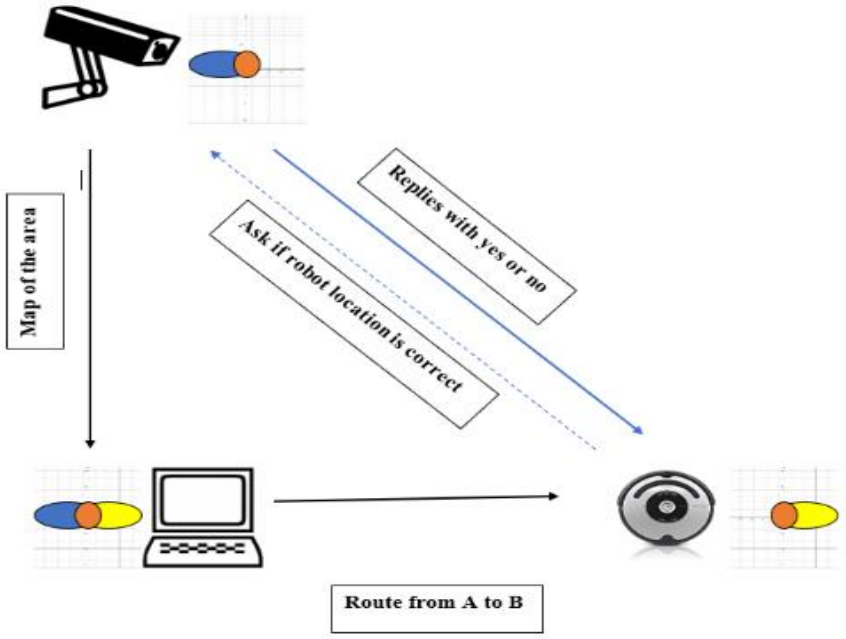

Figure 1: Interaction between parts.

The overhead camera first creates a coordinate plane of the environment. This coordinate plane highlights all edges of every obstacle found within the surrounding area and determines the starting and ending point of both robots. This is done by analyzing the photograph taken of the area. This is then converted to a json file and sent to the algorithm. We chose a json file because it allows us to send information to multiple destinations and is able to convert its data to python objects, giving us easy access to the camera data. Then, a file was generated from the algorithm that contains the original json file and the calculated route for each robot to follow. This file is used by the robot to execute the planned route.

Experimental Approach

To implement this experiment, we needed to do three things: set up the camera, have that camera analyze environment data, and have the robot move based on the path given.

The first thing we decided to do was to set up the camera. The camera needed to see the whole experimental area at an equal perspective, hence, it was essential for it to have a bird's-eye

view. Screwing the camera into the ceiling wouldn't work because of the building structure and the photos we would be able to take would be blocked by the air conditioning piping. It was then decided that we would need to create a physical apparatus that would hold the camera over the AC to get the best possible photo. This was accomplished with a long wooden plank across both AC pipes, creating a bridge that would be secured on both sides with two smaller wooden boards. The camera was then screwed into the center of the long plank. A schematic of this is shown in Figure 2.

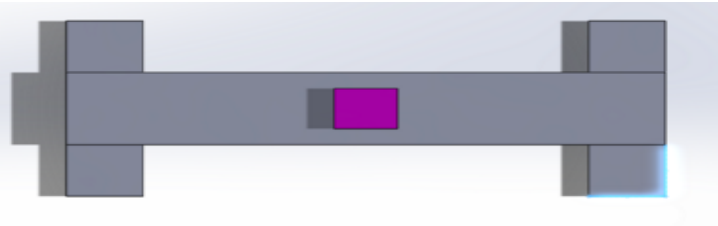

Figure 2: Camera apparatus schematic.

Now that we have the camera physically set up, we can start the process of taking a picture of the experiment and transform this data to a .json file. This file will consist of the coordinates of the obstacles, destinations, and cables. We decided we wanted to use the python package OpenCV [13] because of its built-in features and the extensive documentation. At first, we wanted to use OpenCV and object recognition to get the coordinates of each grouping. However, doing object recognition over a photo is very tedious and it would be hard to differentiate between all our desired objects so it was decided we would try to find a different way to do this with OpenCV. OpenCV itself has a lot of functions that can use colors between a certain color range to find certain objects. If we separate each type of object into a different color grouping, we can find those objects. Green and orange represent the robots and the length of the cable between them, pink shows all potential obstacles, and blue represents the desired destinations. This is shown in Figure 3 below:

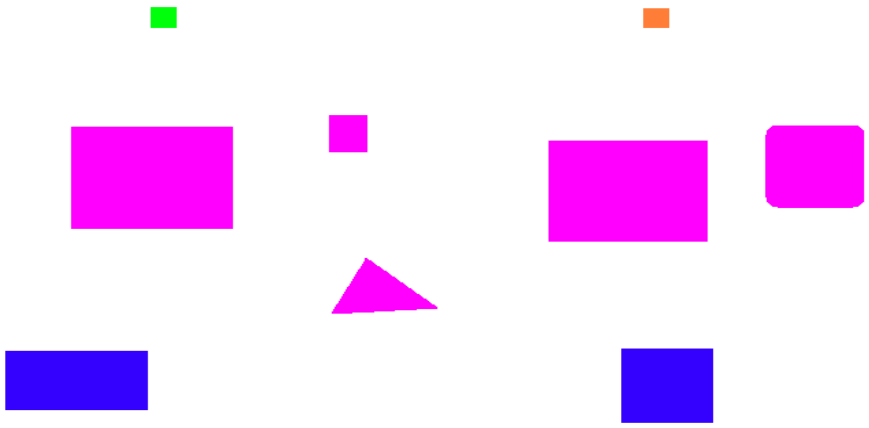

Figure 3: Bird's-eye view of experiment map.

Blue, green, orange, and pink all have difference color ranges, hence, we would get a different pixel array for each one. We then can manipulate these pixels to get the coordinates of the obstacles, destinations, and cables. All this information is then outputted into a .json file which will be used in the algorithm and the robot movement. An example one is shown below:


```
{
    "cable": [
        "132, 66",
        "348, 66"
    ],
    "obstacles": [
        [
            "50, 134",
            "187, 134",
            "50, 229",
            "187, 229"
        ],
        [
            "412, 180",
            "532, 180",
            "412, 272",
            "532, 272"
        ],
        [
            "256, 257",
            "314, 257",
            "256, 304",
            "314, 304"
        ]
    ],
    "destinations": [
        "164",
        "364",
        "484",
        "371"
    ],
    "cableLength": 216
}
```


Figure 4: Example .json file.

Once we finished the camera, we moved to setting up the robots. Our first approach with this step was to take advantage of the open source software, ROS (Robotic Operating System). ROS has many built in packages that work for various robots, including ours. ROS would also be an easy way to do the photo analysis for the obstacles within the area, making connecting the two together quite simple. Once I got the robots equipped with the operating system Ubuntu, I started searching for potential packages that we could use. This led to an endless search of dead-end code. I personally spent months trying package after package to get the robots to move in a straight line. I watched tutorials, wrote personalized code, and emailed a package creator trying to find answers. My biggest issue at that time was that I was quite unfamiliar with open source software and how to capitalize on it. I was used to being able to find some sort of help for each problem that I was given but when a software is open sourced, it can be quite hard to find documentation, let alone a solution. Learning ROS was the biggest learning curve throughout this whole process and if I were to do this project again, I would spend a month or two just trying to learn ROS itself, just so I would be more familiar with the whole ROS process. I was never able to have the robot listen to my commands via ROS, so it was decided that we would find other avenues for both camera analysis and robotic movement.

After searching for different channels of robotic communication, I was able to find a python package, pySerial, that would connect to the robot through serial ports on a computer. The creators of iRobot Create developed code for this package that made this connection work and I was able to get our robots moving in a straight line. Now that the robots could move, I had to use the paths given by Teshnizi's algorithm and correlate it to the coordinates of the cable, obstacles, and destinations given by the camera. The input itself is a 2-D array, as shown here, *[['R1', '00-1','00-2','D1'], ['R2', '01-0', '01-3', 'D2']]*. The first array corresponds to the first

robot and so forth. Once we know which robot that is driving currently, we can start to generate the path. To do this, the code looks at the input with various elements like *'00-1'* or *'01-3'*, which map to the coordinates of the obstacles and destinations. Hence, *'00-1'* would be obstacle 0 at coordinate 1 and *'01-3'* would be obstacle 1 at coordinate 3. This is shown in Figure 5.

```
def createRoute(self):
    global robotRoute
    robotRoute = None
    for i in range(len(robotPath)): #[[R1, 00-1,00-2,D1], [R2, 01-0, 01-3, D2]
        if ( robotPath[i] != "R1" and robotPath[i] != "D1"):
            obtArr = robotPath[i][0] + robotPath[i][1]
            obtArr = int(obtArr)
            obtNum = robotPath[i][3]
            obtNum = int(obtNum)
            cord = inputValues.obstacles[obtArr][obtNum] #.json file input
            cord = cord.split(',')
            cord[0] = int(cord[0])
            cord[1] = int(cord[1])
            robotRoute.addToPath(cord) #path for specific robots
        elif robotPath[i] == "R1":
            cord = str(inputValues.cable[0]) #.json file input
            cord = cord.split(',')
            cord[0] = int(cord[0])
            cord[1] = int(cord[1])
            if not robotRoute:
                robotRoute = Route (robotName,cord)
            else:
                robotRoute.addToPath(cord) #path for specific robots

        elif robotPath[i] == "D1":
            cord = inputValues.destinations[0] #.json file input
            cord = cord.split(',')
            cord[0] = int(cord[0])
            cord[1] = int(cord[1])
            robotRoute.addToPath(cord) #path for specific robots
            robotRoute.setEnd(cord)
```

Figure 5: Creating route code sample.

By finding these coordinates, we can calculate how far the robot would need to travel for each point on their perspective path. I ran the robots multiple times and averaged how long it took to travel a random distance. By using this average, I was able to determine how long it took each robot to travel one foot and how long it took to turn left and right by 90°. With this I could point the robot in the right direction and send it to the correct coordinate. I also wanted to keep track of where the robot was facing to ensure that the robot was traveling in the right direction. I accomplished this by first having the robot be facing 90°. The robot would then turn left or right,

depending on which way they were going and then orient back to 90°. Once the robot believes it is in the correct position, it must check with the camera to make sure it is okay to move. Since we already have a way to pinpoint objects via color, we decided to do it again with this section. The robot will notify the camera that it thinks it is in a desired position and send what position it wants to be in and what position it wants the other robot to be in. The camera then takes a picture of the area and uses the previous color ranges to determine if both robots are near their correct point. If the robot that sent the notification isn't near their point, then it calculates a new length of travel. If the other robot isn't near their point, then the current robot waits until it is. If both robots are correct, then both will start moved to their next point on their path. This will repeat until both robots hit their target. This is process is shown below:

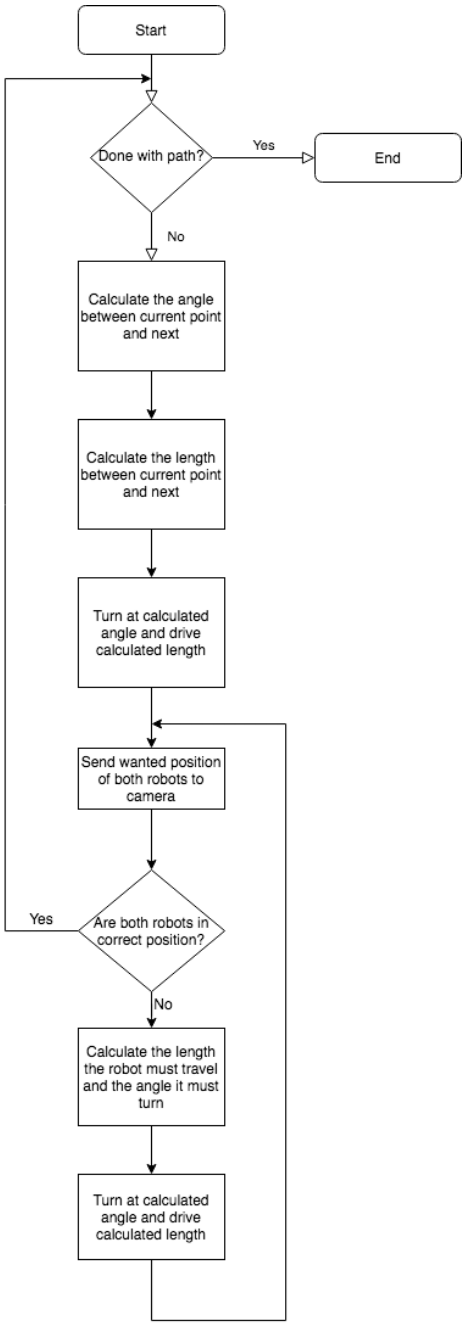

Figure 6: Robot following path flowchart.

Algorithm

The algorithm created by Teshnizi and Shell [5] implements a homotopy cell-based decomposition to determine the fastest route to the given destination. A normal homotopy structure asserts that two different paths are in the same homotopy class if each path can be

deformed to the other. If an object blocks the transformation between two paths, then they are in two different homotopy classes. This is shown in Figure 7 and originally found in [9 Fig. 3].

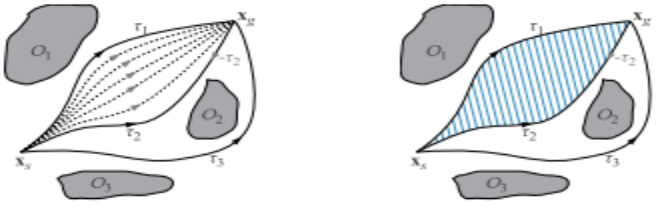

Figure 7: Homotopy Classes

This is useful in determining which homotopy class to take if an obstacle is in the way, however, it doesn't show the shortest route a robot should take nor can each path indicate which homotopy class it is in. This issue can be solved in a cell-based decomposition approach because the authors assert that the homotopy class will be based on the vertices of the obstacle(s) meaning that each path knows which homotopy class it falls in and a shortest path will be guaranteed [7], [8].

In the robot's starting position, it has a travel radius of *l*, where *l* is the length of the cable. Each robot in a certain position can only see the environment not blocked by the obstacle(s) within their travel radius; this is considered the current position's cell. If the destination of a robot is not sensed by the robot, then the robot must go into the area not seen in its current position, traveling to a random vertex *a*, which is a length *A* away from the starting position. However, this causes the cable to shorten to *l* - *A*. Now, the robot's travel radius is shorted, so the new cell of the robot decomposes into a smaller area, with a travel radius of *l* – *A*. This goes on until the destination is reach or the cable becomes too short to move forward. If a route was not found, then it finds a parent vertex that has another child to do the process all over again. This is repeated until the shortest route is found. This is shown in Figure 8 and originally

in [4 Fig. 4].

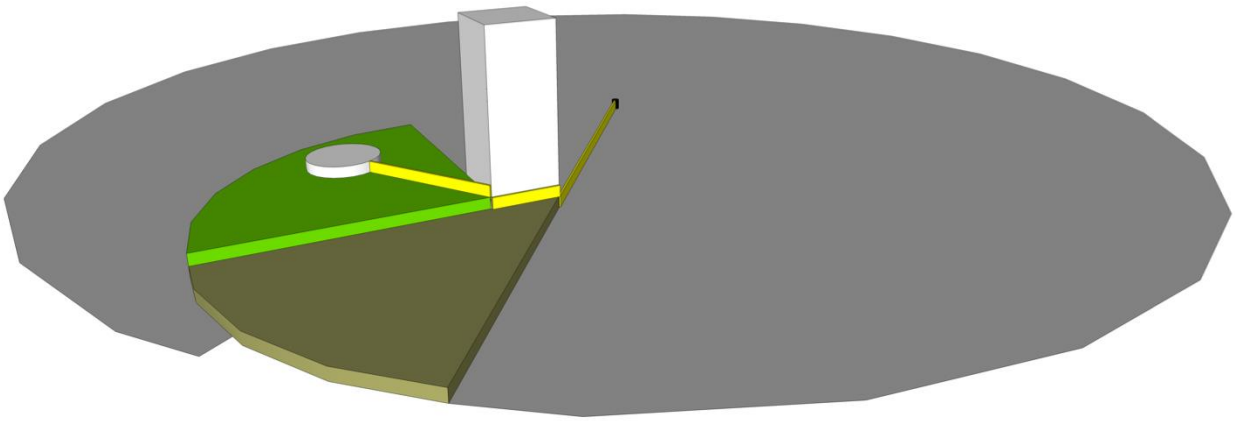

Figure 8: Cell-Decomposition

By using cell-decomposition, we can shorten the number of paths that the algorithm checks, making the algorithm quicker. A quicker algorithm will allow for tethered robots to respond quicker to their environment.

CHAPTER IV

RESULTS

Due to the unforeseeable events of COVID-19, we were unable to accomplish the final goal of testing the algorithm outside of the simulation. We were able to get all parts working individually but were never able to test then together. However, this has allowed us to create a roadmap of each individual piece of the experiment to help continue this experiment for future work. Among the most important parts of the roadmap is the capability to move the robots along a given path and the ability to determine the placement of objects within a photo via color. These accomplishments shown that we have the capacity to finish the experiment at a later time.

CHAPTER V

CONCLUSION

Previous work employs the use of homotopy classes and obstacles; however, the abundance of these classes can make these algorithms run longer to determine the shortest path. This thesis applies an experimental approach to motion planning for tethered robots that depends on cell-decomposition to determine the shortest path. The cell-decomposition uses homotopy classes as a base model but asserts that the use of robot's "eye-sight" will be faster and still develop along the same short path. This concept was first introduced by Teshnizi and Shell [5] and was tested inside of a simulated environment. Our original goal was to test this concept inside of a real-world experiment, however, due to the global pandemic we were only able to create a guideline for future experimentation.

REFERENCES

[1] B. Donald, L. Gariepy, D. Rus, "Distributed manipulation of multiple objects using ropes", *IEEE International Conference on Robotics and Automation (ICRA)*, vol. 1, pp. 450-457, 2000.

[2] S. Kim, S. Bhattacharya, H. K. Heidarsson, G. Sukhatme, V. Kumar, "A topological approach to using cables to separate and manipulate sets of objects", *Robotics: Science and Systems*, 2013.

[3] Bhattacharya, H. Heidarsson, G. Sukhatme, V. Kumar, "Cooperative control of autonomous surface vehicles for oil skimming and cleanup", *Robotics and automation (ICRA) 2011 IEEE international conference on. IEEE*, pp. 2374-2379, 2011.

[4] Reza H. Teshnizi, Dylan A. Shell, "Planning motions for a planar robot attached to a stiff tether", *Robotics and Automation (ICRA) 2016 IEEE International Conference on*, pp. 2759-2766, 2016.

[5] Reza Teshnizi and Dylan Shell, “Computing Cell-Based Decompositions Dynamically for Planning Motions of Tethered Robots,” Texas A&M University, IEEE International Conference on Robotics & Automation, 2014.

[6] S. Lavalle, Planning algorithms, Cambridge University Press, 2006.

[7] Narayanan, Venkatraman, Paul Vernaza, Maxim Likhachev, and Steven M. LaValle. "Planning under topological constraints using beam-graphs." In *2013 IEEE International Conference on Robotics and Automation*, pp. 431-437. IEEE, 2013

[8] E. L. Allgower and K. Georg, “Introduction to Numerical Continuation Methods,” 2003.

[9] S. Hert and V. Lumelsky, "The ties that bind: motion planning for multiple tethered robots," Proceedings of the 1994 IEEE International Conference on Robotics and Automation, San Diego, CA, USA, 1994, pp. 2734-2741 vol.4.

[10] T. Igarashi and M. Stilman, “Homotopic path planning on manifolds for cabled mobile robots,” in Algorithmic Foundations of Robotics IX. Springer, 2011, pp. 1–18.

[11] X. Wang and S. Bhattacharya, "A Topological Approach to Workspace and Motion Planning for a Cable-Controlled Robot in Cluttered Environments," in IEEE Robotics and Automation Letters, vol. 3, no. 3, pp. 2600-2607, July 2018.

[12] D. Grigoriev and A. Slissenko, “Polytime algorithm for the shortest path in a homotopy class amidst semi-algebraic obstacles in the plane,” in Proceedings of the 1998 international symposium on Symbolic and algebraic computation. ACM, 1998, pp. 17–24.

[13] *OpenCV*. (2020). Intel Corporation, Accessed: Feb. 2020. [Online]. Available: https://docs.opencv.org/