LTL Planning for Quadcopter Reconnaissance in Densely Populated Environments

Carson Kohlbrenner, Nolan Stevenson

1) Link to full video of simulated mission here.
2) Link to GitHub repository here.

I. Project Scope

Our project goal was to leverage our knowledge of theoretical temporal motion planning into a realistic system with kinodynamic constraints and error. Before starting this project, our team had no experience using open-source toolboxes such as OMPL, Gazebo, ArduPilot, or ROS. However, these are fundamental and widely used software packages that bridge the gap between theory and application. Thus, throughout this project, we explored how to use these software packages to simulate a theoretical motion plan, and through this learn how to incorporate motion planning into real systems. Our project goals were as follows:

1) Develop an LTL planner using OMPL.
2) Simulate quadcopter dynamics and sensors using ArduCopter autopilot in Gazebo.
3) Autonomously command pathing waypoints to a quadcopter using ROS.

We were able to successfully meet all of these project goals by solving a quadcopter search and rescue problem in a densely populated forest.

II. Introduction

Search and rescue scenarios often occur in densely populated forests where searching for multiple targets on foot can be timely and expensive. With this as our motivation, we present a solution where a remote operator can input a satellite image and then mark regions of interest to search and a specified order to search those regions. A quadcopter motion plan is then automatically generated which visits all necessary regions while avoiding obstacles. This motion plan could then be commanded into a autonomous drone capable of scouting the desired regions quickly and efficiently.

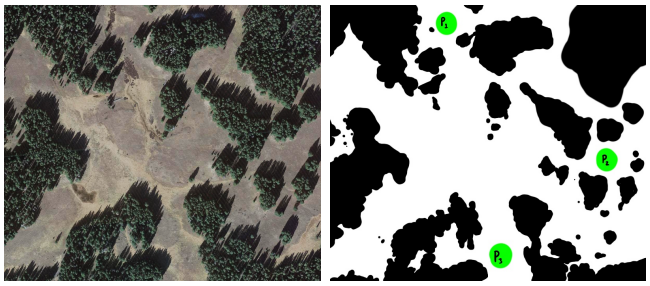
For the motion planner, we decided to use linear temporal logic (LTL) due to its advantages in handling multiple desired goal states over time. LTL is supported by the Open Motion Planning Library (OMPL), so we used this library as the foundation of our planner [1]. To simulate the quadcopter dynamics and sensors we used the open-source platform ArduCopter [2]. This software is typically built directly into commercially available flight controllers through the Pixhawk platform [3]. To communicate between the LTL motion plan and the ArduCopter autopilot, we will be using the Robotics Operating Systems (ROS) as this is one of the most common communication protocols for an autonomous drone [4]. To verify our results, we simulated the drone's hardware using the open-source physics engine Gazebo to visualize the quadcopter's path throughout the mission [5].

All authors are with the University of Colorado Boulder, 1111 Engineering Drive, Boulder, CO USA. name.surname@colorado.edu.

Fig. 1: Using LTL in OMPL, Gazebo, ArduPilot, ROS, and OpenCV, we plan a path for a quadcopter drone to visit multiple points of interest for reconnaissance, scanning, and search and rescue. We demonstrate this using a simulated forest environment where the quadcopter must avoid collisions with trees and other obstacles.

III. LTL Planning in OMPL

The LTL planner we developed for this project generated a series of waypoints for the quadcopter to visit, including selected regions of interest in order of priority. Although the quadcopter's flight will be simulated in $SE(3)$, we constrained this problem to $SE(2)$ by specifying a **constant altitude of 3.5 meters** for the quadcopter to fly at. The benefit of using OMPL as the backbone of our LTL planner is that it already contains a synergistic planner that will solve paths for a specified deterministic finite automata (DFA), abstraction graph, and low-level planner. We decided that a grid-based abstraction level, which breaks the workspace up into discrete square cells, most intuitively encompasses the ability of a remote operator to select regions of interest. To set up this abstraction layer, we created a new decomposition class, called `MyPropositionalDecomposition` that inherits from OMPL's `PropositionalDecomposition` and `GridDecomposition` classes. In our custom class, we



(a) (b)

Fig. 2: Satellite images can be used for finding the mission bounds and obstacles to avoid. a) Sample satellite image of a forest [7] b) Conversion of the original image where white signifies the mission zone, black is a no-fly zone, and the green dots are regions of interest denoted as p_1, p_2, p_3 in order of priority.

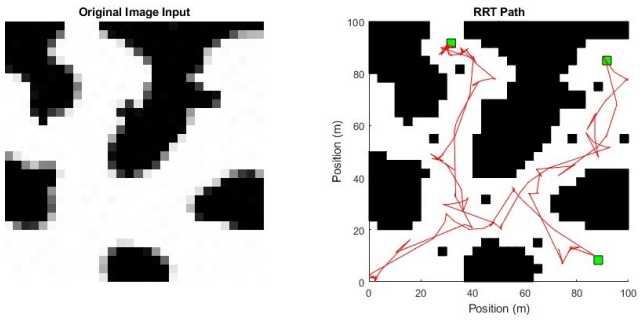


Fig. 3: Conversion between a greyscale image to a motion plan. Note, additionally randomly-distributed obstacles were added in post process to increase the path planning complexity.

can specify the bounds, dimensions, and number of grid cells of the workspace. After defining the size of the workspace, the custom class assigns a unique *Region ID* (RID) to each cell.

A. Image to Workspace

We wanted the remote operator to be able to input an image of the search area, whether it came from a quadcopter or satellite, and quickly get a scoped workspace with obstacles. For this project, we manually converted a satellite image to a monochrome format where black regions indicate the locations of trees. Fig. 2 demonstrates the process of converting a satellite image to a workspace. For computational purposes, we used a simplified lower resolution version of Fig. 2 as shown in Fig. 3 for the remainder of this project. To convert the forest image as seen in Fig. 3 to a usable workspace, we passed in the monochrome image as an OpenCV `Mat` object to the planner [6]. The planner then read pixel data about the image and adjusted the workspace grid to have as many grid cells as pixels. Next, it calls Algorithm 1 to read each pixel's color value in the range of $0-255$ where 0 is black and 255 is white. If any pixel has a color reading below 250, the pixel's associated *RID* is stored as an obstacle in the `MyPropositionalDecomposition` object's *RID* list. This *RID* can then be used to check for collisions when propagating the low level planner.

B. Abstraction and Deterministic Finite Automata

Similar to how the obstacles are assigned, regions of interest, or propositions, were also kept track of in a list in the `MyPropositionalDecomposition` object. The user has to specify *RIDs* for the quadcopter to visit and they get wrapped and assigned to a proposition p_m where m is the number of propositions at the time of assignment. To keep the automata co-safe, the LTL function was described in sequential order as followed and takes the form of Fig. 8.

$$F(p_{\text{takeoff}} \wedge F(p_1 \wedge F(p_2 \wedge ...F(p_m \wedge F(p_{\text{land}}))))) \quad (1)$$

This says that the quadcopter shall eventually takeoff, visit each of the regions of interest in order or prioirty, and land in that order. The product of the automata in Fig. 8 and the abstraction grid layer linked to the propositions was then produced using a built-in OMPL `ProductGraph` constructor where it is ready for motion planning using the built-in SyCLoP planning scheme.

C. Low Level Planner

The low-level planner chosen for this project was RRT due to its ability to quickly converge to a goal. The idea behind the LTL planner is to be agnostic to multiple quadcopter configurations, so calculating for kinodynamic constraints in the low-level planner would have been restricting. Instead, we specify a controlled yaw limit of $\pm\frac{\pi}{8}$ radians when propagating the tree to avoid sudden turns. Additionally, without incorporating the dynamics of the quadcopter into the motion planning, we needed to implement additional safety measures to ensure the quadcopter won't collide with the environment during simulation. The size of the workspace used in this paper was $100\times100m^2$. Thus, we can assume with reasonable confidence that a small quadcopter's dynamic constraints will cause only small deviations from the specified path due to the scale of the path. To combat this deviation, we implemented a conservative configuration space restriction by taking the Minkowski difference of the obstacles and quadcopter, assuming that the quadcopter was the size of a full grid space ($11.1m^2$ in this case). This resulted in a path that allowed for slight deviations. Fig. 3 demonstrates an example of an RRT sampled path created by the low-level planner using the bounds of a simplified satellite image.

A limiting factor of the RRT-generated path is that multiple waypoints would frequently appear clumped to each other. Due to the nature of the flight controller as explained in Section IV, clumped waypoints would cause the quadcopter to make sharp, jagged movements and frequent stops. To solve this problem, we implemented Algorithm 2, a *waypoint region fit algorithm* that iterates through a path and removes waypoints in the same grid cell as the previous waypoint. This function more than halved the number of waypoints of the path shown in Fig. 3 ($197 \rightarrow 94$ waypoints). This allowed the quadcopter to motion plan through fewer waypoints which allowed a smoother, more continuous path without sacrificing visiting regions of interest. The effect of this algorithm on the RRT generated path can be seen in Fig. 6

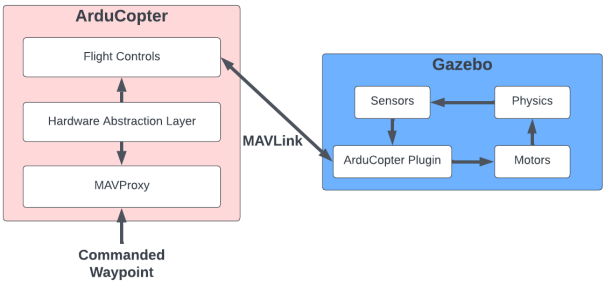


Fig. 4: Flowchart of interactivity between ArduCopter and Gazebo. A commanded waypoint is sent to ArduCopter and converted to flight controls before being sent to the drone in gazebo.

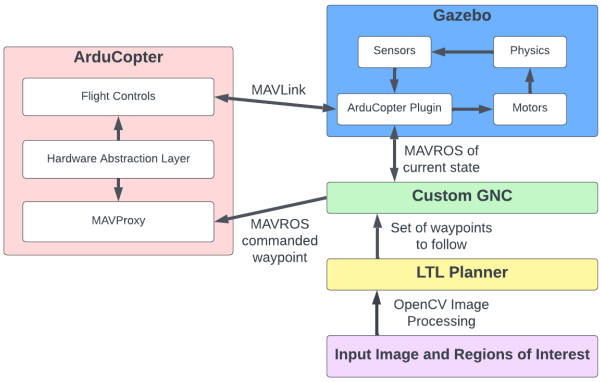


Fig. 5: Flow chart for autonomous control of quadcopter

IV. Quadcopter Dynamics using ArduCopter in Gazebo

To incorporate and simulate quadcopter kinodynamic constraints, we utilized a physics engine coupled with an autopilot software. The physics engine we used was Gazebo, a widely used advanced robot simulator capable of emulating physical collisions, sensor error, and more. The autopilot software we used was ArduPilot Copter (ArduCopter), an open-source multicopter UAV controller and simulator. These software applications can be coupled together using a communication protocol called MAVLink, micro air vehicle link, a protocol for translating inertial commands into vehicle thrust commands [8].

To implement a quadcopter representation in the Gazebo environment, we utilized the 3DR Iris Quadrotor Gazebo model file produced by PX4 autopilot [9]. By importing this model into a empty Gazebo world file, we were able to generate a visual representation of a quadcopter.

Our next step was to establish a MAVLink connection between the quadcopter and an emulated firmware onboard the quadcopter. To do this, the ArduCopter firmware was incorporated as a plugin to the Gazebo model file. This then allowed the ArduCopter firmware to start upon Gazebo simulation startup. We were then able to connect to the firmware by establishing a MAVLink connection with the ArduCopter plugin. This allowed the ArduCopter software to access the physics and sensors present in the Gazebo simulation. By making this connection between softwares, the ArduCopter autopilot was no longer running on perfect sensor data and was instead using realistic data collected from IMU and GPS systems within the Gazebo simulation. This fulfills the goal of this project, which is to simulate a realistic system completing a motion planning problem.

To fly the quadcopter in Gazebo, MAVLink commands were sent to MAVProxy to control the quadcopter to commanded inertial waypoints. MAVProxy software is able to communicate directly with the ArduCopter flight controls through an abstraction layer to fly the quadcopter. The abstraction layer leverages the physical stats of the quadcopter to translate the commanded waypoint to a commanded set of motor torques. A flowchart depicting this relationship can be seen in Fig. 4. This protocol established the baseline for how we were able to control a quadcopter in a Gazebo environment to follow a waypoint path.

V. ROS Communication between Planner and ArduCopter for Autonomous Command

To *autonomously* control the quadcopter in the Gazebo simulation, we utilized a combination of open-source quadcopter functions and custom functions to create a Guidance, Navigation, and Control (GNC) script to iteratively feed the autopilot inertial x, y, and yaw commands throughout the mission [10]. In our custom GNC function, we specified $[x, y, \psi]$ waypoints for the quadcopter to fly to. Once the quadcopter was within a specified **1.5 meters** (approximately half a grid space) ϵ tolerance of a waypoint in the list, a new waypoint was commanded. Additionally, we wanted to have the quadcopter's front-facing camera see the regions of interest upon approach, so we command the desired yaw of the quadcopter to always point the camera in the direction of quadcopter motion. This was an iterative process that kept feeding in new waypoints to the quadcopter until it was finally commanded to land.

In order to identify when the quadcopter is within a certain tolerance of an inertial waypoint, the state data of the quadcopter's position in Gazebo must be examined. Using MAVROS, a ROS plugin that we ran alongside ArduCopter, we were able to continuously publish the output state estimation of the quadcopter. Then, the state estimate from the GPS and IMU data was leveraged by the GNC script to decide when a waypoint is reached and motion plan through the waypoint path.

To prompt the ArduCopter autopilot with waypoints, MAVROS was used to send commands to MAVProxy which could directly control the ArduCopter controls. Using this method, we bypassed the need for manual inputs and autonomously commanded from a set of waypoints stored in a sequential list from the LTL planner. The flowchart for this process can be seen in Fig. 5.

VI. Results

To visualize the results, we created a Gazebo world environment that matched the example image used by the

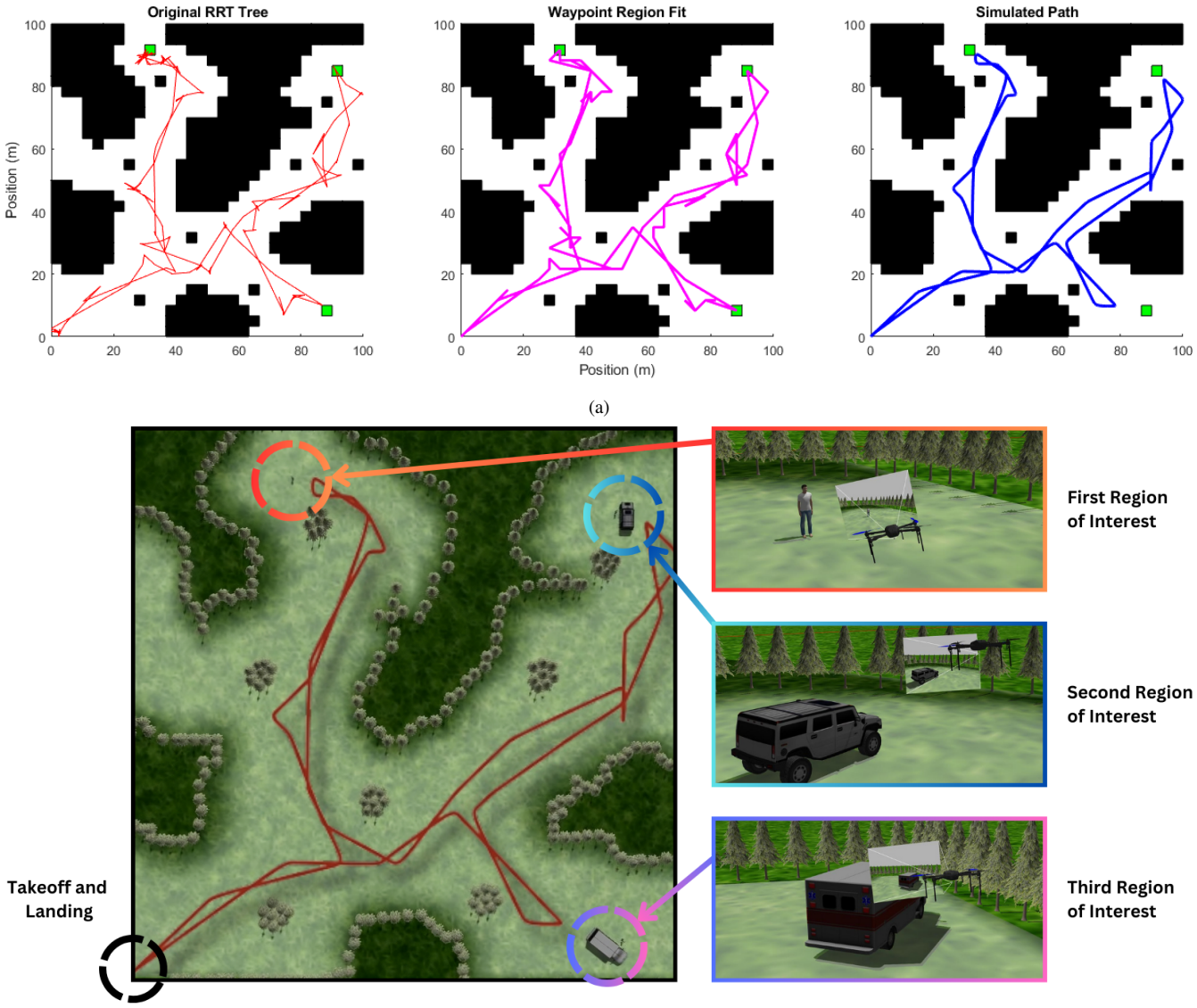


Fig. 6: a) Simulated flight path comparing the original RRT path planned with LTL to the waypoint fixed graph restricting waypoints to grid spaces and the final simulated path with quadcopter dynamics. b) Visualization of the simulated drone reaching the regions of interest.

(a) (b)

Fig. 7: a) Monochrome satellite image. b) Gazebo replica of satellite image.

LTL planner (see Fig. 7. This was done by applying the image as a texture on the ground model in Gazebo. Then, tree obstacles were placed over all black regions, and regions of interest were placed in the green regions from Fig. 3. Specifically, a human was placed in interest region one, a jeep was placed in interest region two, and an ambulance was placed in interest region three. The resulting Gazebo world file can be seen in Fig. 7 This allowed us to validate that the quadcopter did indeed go to all regions of interest before landing.

With the world file complete, the mission was simulated using the software flow down described in Fig. 5. The resulting path can be seen in Fig. 6 and the resulting video can be seen here. The quadrotor was able to successfully reach each region of interest in the $10,000\ m^2$ area without

colliding with obstacles in **2 minutes and 9 seconds**. Fig. 6 shows the final full path of the mission.

Analyzing the results, the simulated path of the quadcopter is much smoother than the original RRT and the waypoint fit path. Through testing, we found that a larger tolerance increased the speed and smoothness of the quadcopter's path; making the motion of the quadcopter appear more realistic to human command. However, greater waypoint tolerances allowed the quadcopter to vary more from the calculated path, which could be an issue in highly contested environments. Because we applied the constraint that neighboring cells of obstacles are also obstacles to the low-level LTL planner, our quadcopter is still able to avoid obstacles and complete the mission while smoothing the path.

VII. Conclusion

In conclusion, our project set out to leverage motion planning theory into a realistic system with kinodynamic constraints and error. Despite our initial lack of experience with open-source toolboxes such as OMPL, Gazebo, ArduCopter, and ROS, we successfully achieved our project goals.

Our approach addressed a significant real-world problem by employing a quadcopter with path-planning capabilities for search and rescue scenarios in densely populated forests. Utilizing linear temporal logic (LTL) and the Open Motion Planning Library (OMPL), we generated a series of waypoints for the drone, prioritizing specific regions of interest while avoiding obstacles.

The integration of ArduCopter in Gazebo provided a realistic simulation environment for quadcopter kinodynamic constraints, and the use of ROS facilitated communication between the LTL planner and the ArduCopter autopilot, enabling autonomous control. By incorporating open-source tools and custom functions, we successfully demonstrated the feasibility of our system in a simulated forest environment.

Our results showcased the effectiveness of the system, as the quadcopter followed the planned path, visiting multiple points of interest for reconnaissance, scanning, and search and rescue without collisions. The smoother trajectory achieved through tolerance adjustments enhanced the realism of the quadcopter's motion while the extra safety factor on the LTL low-level planner ensured obstacle avoidance.

In summary, our project not only achieved its goals but also demonstrated the potential of integrating motion planning theory into practical applications, highlighting the value of open-source tools in bridging the gap between theoretical concepts and real-world implementation. This work opens avenues for further exploration and development in the field of autonomous systems for complex scenarios like search and rescue missions.

References

[1] I. A. Sucan, M. Moll, and L. E. Kavraki, "The open motion planning library," *IEEE Robotics & Automation Magazine*, vol. 19, no. 4, pp. 72–82, 2012.

[2] "Arducopter," https://ardupilot.org/copter/, [Software].

[3] "Pixhawk flight controller," https://pixhawk.org/, [Online; accessed December 2023].

[4] "Robotics operating system (ros)," https://www.ros.org/, [Software].

[5] "Gazebo," http://gazebosim.org, [Software].

[6] "Opencv library," https://opencv.org/, [Software].

[7] "Google earth satellite image," https://earth.google.com/web/@39.14882051,-107.74775177,3184.25387738a,1433.65263345d,35y,0h,0t,0r/data=OgMKATA, [Online; accessed December 2023].

[8] "Mavlink communications software," [Software].

[9] "Iris gazebo model," [Software].

[10] "Basic gazebo gnc tutorials," https://github.com/Intelligent-Quads, [Online; accessed December 2023].

[11] "Spot ltl visualizer," https://spot.lre.epita.fr/, [Online; accessed December 2023].

APPENDIX

A. Figures:

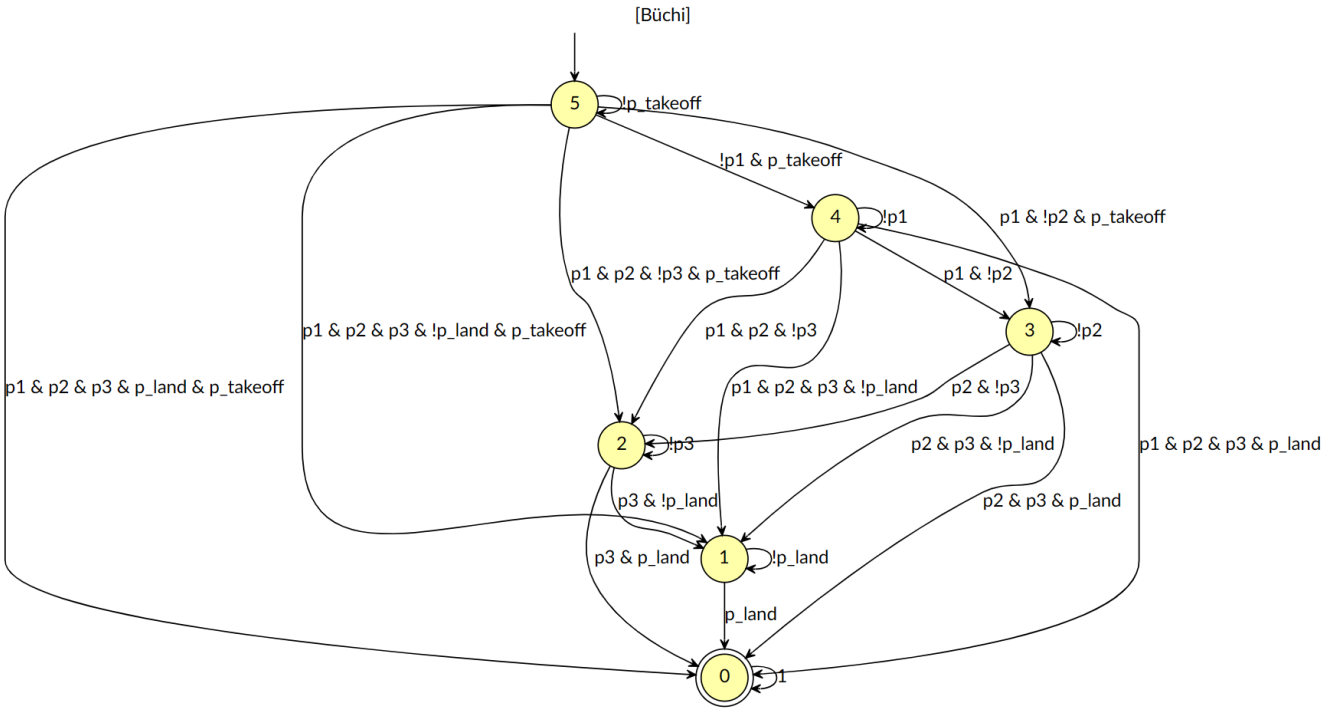


Fig. 8: Deterministic Finite Büchi Automata for taking off, visiting three regions interest, and landing [11].

B. Algorithms:

Algorithm 1 Add Obstacles from Image

procedure ADDIMAGEOBSTACLES($MyPropositionalDecomposition, image$)
  for $i \leftarrow 0$ **to** $image.rows$ **do**
    for $j \leftarrow 0$ **to** $image.cols$ **do**
      if $image(i, j).color < 250$ **then** ▷ Pixel is not white
        $x_{coord} \leftarrow i/image.cols$
        $y_{coord} \leftarrow j/image.rows$
        $rid \leftarrow$ RID at location (x_{coord}, y_{coord}) ▷ Get the region ID
        if rid = start state or neighbor of start state **then**
          Continue
        else
          Add obstacle to $MyPropositionalDecomposition$ object
        end if
      end if
    end for
  end for
end procedure

Algorithm 2 Fit Path Points to Grid

procedure GRIDFIT($decomp, boundMax, gridLength, stateSpace$)
    $previousRegionID \leftarrow -1$
    $previousCoordinates \leftarrow [0, 0]$
    $previousDirection \leftarrow [-10, 10]$
    $repeatCount \leftarrow 0$
    Open "pathGrid.txt" for writing as $outputFile$
    Open "pathRaw.txt" for reading as $inputFile$
    while there are lines in $inputFile$ **do**
        Read a line from $inputFile$ and split into $xStr, yStr, yawStr$
        Convert $xStr, yStr, yawStr$ to doubles: x, y, yaw
        Create a new state in $stateSpace$ with x, y, yaw
        $regionID \leftarrow$ Locate the region of the state in $decomp$
        if $regionID$ is same as $previousRegionID$ **then** ▷ Drone is in the same grid cell.
            continue to the next iteration ▷ Don't store waypoint, move to next one.
        end if
        $gridCoordinates \leftarrow$ Convert $regionID$ to grid coordinates
        $scaleFactor \leftarrow boundMax/gridLength$
        $currentDirection \leftarrow$ Difference between $gridCoordinates$ and $previousCoordinates$
        if $currentDirection$ is same as $previousDirection$ **then** ▷ Drone is going the same direction.
            Update $previousCoordinates$ and increment $repeatCount$ ▷ Don't store waypoint, move to next one.
            continue to the next iteration
        end if
        if $repeatCount > 0$ **then** ▷ Drone is in a new grid cell going a new direction.
            Write the midpoint of the previous grid cell to $outputFile$ ▷ Store waypoint.
            Reset $repeatCount$ to 0
        end if
        Write the midpoint of the current grid cell to $outputFile$
        Update $previousCoordinates, previousRegionID, previousDirection$
    end while
    Write "0 0 0" to $outputFile$ to mark the end
    Close $outputFile$ and $inputFile$
end procedure