Sudikoff Sentinel: A Seek and Destroy Robot

Pratinav Bagla, Isaac Feldman, Sherry Liu and Elliot Potter

***Abstract*— Patrolling a known space to detect intruders is a coverage problem that requires efficient sparse graph creation, planning, traversal and object detection. We propose a Seek and Destroy robot: the Sudikoff Sentinel. When given an accurate occupancy grid, the Sudikoff Sentinel will localize, construct a sparse Voronoi graph, and find an optimal closed-loop path over all edges in the graph. It then patrols the route until it finds an anomalous object in its LIDAR scan, at which point it routes toward the object using a camera and PID controller, and "destroys" the object. It then returns to the graph and resumes patrolling along the path.**

I. INTRODUCTION

Malicious students roam the hallways of Sudikoff, causing various distractions and mischief. Sudikoff would be a safer and more productive place if these individuals could be chased away. Patrolling Sudikoff requires efficiently traversing a known area, and seeking and destroying any detected intruders.

A. Prior Work

1) Patrol Planning: Patrol Planning is a subset of path planning where the goal is to produce a path that takes the agent through a space in a complete manner. It is a type of coverage problem. In the state-of-the-art there are two main methods of producing these plans: learning based methods and methods derived from Generalized Voronoi Diagrams (GVD). What follows is a short summary of these methods and their application to patrol planning.

a) Learning-based Methods: Learning based methods employ some form of feature extraction on the input occupancy grid map and then set up a topological graph to be optimized. The goal of these maps in addition to the paths created, are to capture the natural structure of built environments in the maps (i.e. each room is a logical cell for the robot to traverse) or even encode semantic information into the cells (i.e. "this room is the kitchen"). One method uses learning combined with input features are derived from a Voronoi graph [1]. In a later method they identify corner features in the input map and use them to create candidate "doors" that divide the map into cells [2]. These cells are then optimized to find a topological graph using a genetic algorithm. Classifier approaches as also common: they sort each point of the occupancy grid into a semantic category ("hallway", "room") and then use these to probabilistically divide the map into cells for the final topological map [3]. The main drawback of these methods is their higher computational cost. While robot on-board processing power has increased, training classifiers and running optimization algorithms are better suited for offline pre-processing. While this is not a problem in itself, for our purposes this may be the wrong choice.

b) Generalized Voronoi Diagrams: Generalized Voronoi Diagrams are a computational method that mark edges in the map such that the points on those edges are equidistant from obstacles. If the robot were to follow such a path, it would avoid all obstacles while traversing the entire reachable area of the map. Methods in the past decades have focused on the efficient construction of these graphs and optimizing their performance in noisy environments. One method uses the BrushFire pixel expansion technique to generate the graph and then optimizes it using a thinning algorithm [4]. Another devises a method for online creation of a Generalized Voronoi Graph (GVG) while the robot moves in the environment [5]. If the robot takes care to stay away from obstacles with a control law, its path can be optimized into a GVG. Finally, this recent method uses a technique that could be compared to GVD [6]. Using image processing algorithms, they skeletonize the map and produce nearly noise-free paths even with noisy maps. The downside is that the running time is very long. Overall, the GVD approach is common and has many fast algorithms for its implementation. Its drawback is that it is not very resilient to noise and does not promise a smooth or complete path through the environment.

c) Traversal of Generalized Voronoi Diagrams: Finding efficient traversal of Voronoi diagrams may be done with several different algorithms, some of which are discussed in [7]. These algorithms vary by computation speed and optimality of the resulting graph. The most optimal solutions produce an Eulerian circuit from a graph from only even-degree nodes. These circuits are closed loops that cover every edge of a graph exactly once. Yamen et al.'s algorithm and Hierholzer's Method are two different methods that produce such circuits. Both algorithms will produce Eulerian circuits from Eulerian graphs. Graphs that contain odd-degree nodes must be converted into ones with exclusively even-degree nodes in order for Hierholzer's and Yamen's methods to work on them. Charpentier describes an algorithm to convert a graph containing odd-degree nodes into one containing only even-degree nodes [8]. This method involves finding pairs of odd-degree nodes, such that the summed cost of the paths between the nodes are eliminated. At this point, additional nodes and edges are added to convert the odd nodes to even nodes.

B. Proposed Solution

We propose a Seek and Destroy robot, which, when given an accurate occupancy grid[1] , will localize, construct a sparse Voronoi graph, and find an optimal closed-loop path over all edges in the graph. It then patrols the route until it finds an anomalous object in its LIDAR scan, at which point it routes toward the object using a camera and PD controller, and "destroys" the object. It then returns to the graph and resumes patrolling.

II. METHOD

A. Assumptions

Our project uses the Husarion rosbot, an autonomous robot equipped with an RPLIDAR scanner and an RGBD camera which moves in a 2D environment. When the robot identifies moving objects, it will pursue the target until the environment is clear, then resume its patrol along the graph.

We assume that moving objects will move away from the robot once they are being actively pursued, that the robot's camera captures video at a high enough FPS to not lose the target between frames, and that the static elements in the environment like furniture will not be changed after the initial map is given to the robot.

B. Localization

The localization module localizes the robot in its environment using an occupancy grid of the environment and LIDAR scan data.

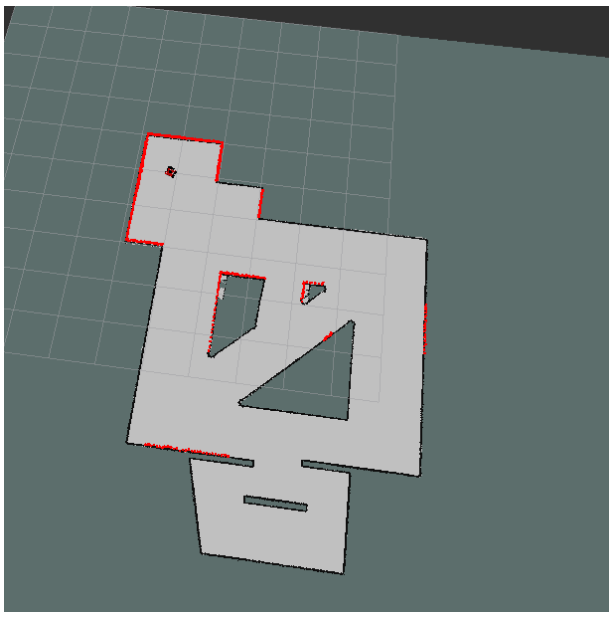

Fig. 1. The occupancy grid created by the patrolling robot. The grid is created before running the program.

[1]We constructed the occupancy grid by cloning, installing, building and running gmapping from `https://github.com/husarion/rosbot_description`. We then ran the patroller module so that the robot would be driven over the entire environment. Finally, we ran `https://github.com/isaac-400/cs81-final/blob/main/src/utilities/write_map_to_file.py` to write the map to a file for future usage.

First, the module computes the distance to the nearest wall for every cell in the occupancy grid in directions: forward/north, backward/south, left/west, and right/east. These data points are compared to LIDAR scan data to determine the probability of the robot being at a particular position.

The initial belief state of the robot's location considers all empty locations in the grid in all possible orientations. The module iterates over every possible pose (x, y, orientation), comparing the LIDAR scan of the distance to the nearest wall in the four directions to the occupancy grid data. The error between these is measured by squaring the difference in each direction and summing them up, giving us a sum of squared errors for each pose.

Since, for our purposes, we do not need to know the entire probability distribution of the robot's state, but only the pose(s) that are highly likely, we eliminate the poses that have an error above a threshold level that makes the pose implausible.

Once we obtain a set of plausible poses, we refine this further by moving the robot one grid square forward (or rotating if there is no space), and then iterating over the set of poses, repeating the same process of comparing LIDAR data and occupancy grid data, and eliminating the poses with implausibly high errors.

Finally, with enough repetition, we should obtain a single pose that the robot has the highest likelihood of being at.

C. Voronoi Graph

Taking inspiration from EVG-thin, we use mathematical morphology methods to compute an approximation of the Voronoi graph for path planning purposes [9]. The goal is to determine the free configuration space for the robot on the map and then devise an efficient path though that space by extracting its skeleton. We import the occupancy grid map from its ROS topic, expand its edges by the robot's width to get the free configuration space and compute the Euclidean distance transform of it. When we thin the Euclidean distance map, we're left with an approximation of the boundaries of the Voronoi cells where the feature points are the walls and obstacles. We extract the key points from this thinned boundary and turn it into a graph. We used the Scipy library's implementations of dilating and skeletonizing the map along with scikit-image library's corner extraction implementation [10], [11]. At this point, the Voronoi graph is mostly constructed, but must be modified in order to be traversable. Many nodes often end up being clustered in small areas of the graph. We thin these by iterating through nodes and deleting any that are within a radius of 1 meter of the selected node.

D. Patrolling

Our patrolling module solves an adapted version of the Chinese Postman Problem. We find an optimal closed-loop path that takes the robot over all edges in the graph. This allows us to perform the computation for the graph once, and then re-use it for patrolling later. This works especially well when the robot seeks and destroys a target, then returns to

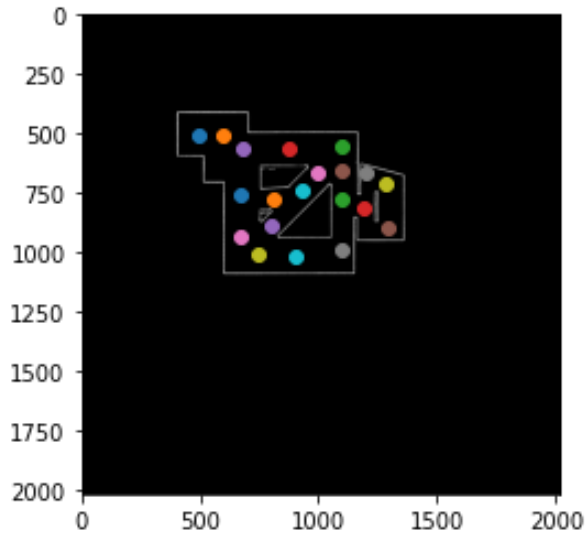

Fig. 2. Visualization of an example Voronoi graph

the graph and can continue to use the previously-computed patrol route. When the robot completes a loop over the graph, it will be at the starting position again, and will simply re-loop.

Our algorithm is broken into the following steps:

1) Find all odd-degree nodes in the graph. The cost of this step is O(1).
2) Calculate the distance across the graph between all pairs of nodes, using Dijkstra's algorithm. Using our implementation of a priority queue (a list), this costs [12]
$$O(V_{odd}^2 \times E \times V) \quad (1)$$
.
Where V_{odd} is the number of odd vertices, E is the number of edges and V is the total number of vertices. This can be improved trivially by using a heap to
$$O(V_{odd}^2 \times E \times log(V)) \quad (2)$$
and can be further improved (though not necessarily changing the worst-case efficiency) by implementing an efficient caching policy of distance between nodes.
3) Find the minimum-cost solution such that all odd-degree nodes are in pairs. This is a recursive solution that costs $O(V_{odd}!)$; i.e., the factorial of the number of odd vertices. An efficient caching policy that stores cost keyed by the set of previously consumed vertices would only require re-computation for a different set of previously consumed vertices, which would significantly improve efficiency.
4) Add fake nodes between the optimal pairs (Use Dijkstra's again to find the optimal paths). If the pairs have no nodes between them, insert a single fake node connected with both parent nodes. Otherwise, create fake nodes in the same locations on the graph as the nodes connecting the parents. Keep connections only on the optimal path found between the parents. At this point, the graph (including the fake nodes and edges) contains only even nodes.
5) Use Hierholzer's Method to find a minimum-cost closed loop over all edges in linear time [7].
6) Either remove the fake nodes or convert them into their real counterparts. This yields the shortest path over a graph of nodes with arbitrary degree.

E. Restoring to Graph

We use an adapted version of breadth-first-search (BFS) where the search completes once any node in the graph is reached. Once this occurs, we perform a ray-tracing operation that eliminates unneeded intermediate nodes in the path found by BFS.

For every node in the path, starting from one extreme end, we check if the other extreme end is accessible. If not, we will move the first evaluated end toward the other. Once we find a clear path, we will move one of the extreme ends to the less extreme point of the evaluated line segment.

We will do this iteratively until the extreme ends are connected with sparse lines.

F. Object Detection

The robot uses both the LIDAR and the RGB camera to detect objects. It uses LIDAR to detect intruders when patrolling, and the RGB camera to follow intruders when chasing.

As the robot patrols the environment, the LIDAR will continuously scan the environment, using the odom to transform its current position into the occupancy grid's reference frame. It will use the LIDAR ranges to find the distance of objects around it.

For the LIDAR to consider something an intruder, it has to meet two criteria:

1) There must be something detected in the LIDAR's range that is not recorded in the occupancy grid.
2) The obstacle must consist of at least 3 consecutive unexpected ranges in the LIDAR readings. The amount of 3 readings was determined by the function of the robot. This robot is intended to be deployed in Sudikoff, where it will chase students' ankles. The average ankle is 11 cm in diameter. Constructing an isosceles triangle with the maximum distance of 1.5 meters as the legs and a base angle of the size of LIDAR's `angle_increment`, we are able to find that the base of the triangle to be a little more than 10 cm, or 3 readings. Any smaller and we consider the object too far to be detected.

When the LIDAR finds something in range that is not on the occupancy grid, it will publish the string "chaser" in `mode`, a topic that controls the current state of the robot. It will also publish a float32 number in the `angle` topic, which will be used to rotate the robot to the initial position it needs to be to chase the intruder.

G. Pursuit

Once the robot is in "chaser" mode, it will track use the `camera_detect` node and the `camera_detect` node to detect the intruding obstacle. Both of these nodes only run when the state is chasing.

In chaser mode, the robot's RGB camera is used to track the intruder object. Using the OpenCV library, the camera detects the BGR (blue-green-red) color range for the orange obstacles and creates a mask based on the values which fall in the range. We then check to see if the mask found enough pixels that matched the target orange color. If it did, it will use the leftmost and rightmost values in the mask to find the midpoint of the shape.

Fig. 3. The primary invader obstacle used to attract the robot

The RGB camera detector publishes two topics. First, it will publish the mode. If it was able to identify enough colored pixels in the color mask, it will broadcast "chaser" into the mode topic, telling the robot to stay in the chasing state. Otherwise, it will publish "restoring" in the mode topic, which tells the robot to return to the graph and stop chasing.

The RGB camera detector also publishes a topic called "angle". The camera has a measured angle of view of approximately 90 degrees. Using the angle of view and midpoint calculated from the layer mask, the camera node will find the approximate angle of the midpoint and determine an angle to rotate to the left or right. The calculated angle is published as a float32 number, and is only used by the PID in chasing mode.

The chaser node is a PID controller that uses the angle of the object published by the camera to determine the change in the robot's velocity. For the P component, the node just uses the angle broadcast. In the D component, the node uses the difference between the current published angle and the previously published angle to find the error between how much the robot actually turned and how much it should have turned. The I component finds the sum of all components. Through testing and fine-tuning, we found that high KP and KD coefficients and a low I coefficient produced the best results when following a moving target.

III. EVALUATION

We performed the following 5 experiments on the robot to test different aspects of the robot's behaviour, as well as to make sure that it was performing in the right way during development.

- *Empty environment:* Path taken while patrolling around empty environment
  - Our first test was to make the robot patrol the empty map from start to finish, completing several loops of the path generated by the Voronoi graph.
  - We had two main characteristics to test:
    - Coverage: Did the robot and its line of sight cover the entire graph? We found that the robot was able to patrol the entire graph and cover all possible places where there might be an intruder.
    - Efficiency: Did the robot traverse the graph quickly while minimizing repeated routes? We found that the robot's path was the most efficient out of the two options tested: our path planning and random walk. Our robot entirely covered the graph in 5 minutes and 10 seconds; a comparable random walk took 15 minutes, 30 seconds and failed to explore the center of the map and the room at the bottom.
- *Static:* Insert a static obstacle
  - After making sure that the robot was patrolling the entire environment efficiently, we tested its reaction on a static obstacle, an orange traffic cone. We put the traffic cone in different parts of the graph out of the view of the robot and would wait until it came into its line of sight, visualized through rqt. We would then wait to make sure the robot was using the PID to route toward the intruder. We found this occurred reliably.
- *Disappearing:* Delete obstacle after robot has started tracking
  - Once we confirmed the robot was able to track and approach obstacles, we then deleted the obstacle after the robot was within a certain range. This simulated the obstacle moving out of sight and was meant to test the robot's ability to restore itself to the patrol route.
- *Moving:* Move obstacle around in and out of view
  - Next, we tested the robot's ability to chase a moving object. We simulated this by putting an obstacle in a position where the robot would eventually detect it in its line of sight, then dragging the object to different positions. We tested both the scenarios when the object remained in the line of sight, allowing the robot to chase it, and also the scenario where the object disappeared from its line

of sight, mimicking an intruder that was chased away by the robot.

- *Lighting:* Different colored obstacles to simulate different lighting conditions
  - Finally, we tested the robot's ability to detect a range of colors. We used different obstacles in the shades red, yellow, and brown, to mimic the color values that the robot's RGB camera would detect in situations with harsh or low light. We then fine-tuned the RGB values for the camera's detection, so that it would be able to function in a variety of environments. We tested traffic cones, bookshelves and a dumpster. The robot detected all these obstacles. (The robot had more trouble with cardboard boxes and car tires).

IV. CONCLUSIONS

In general, our robot accomplished the goals we set for it. It is able to patrol a mapped environment efficiently and quickly, can deviate from the path to chase intruders, and restore itself to the path accurately.Our robot also succeeded in all of our testing metrics. It is around 300 percent faster than a random walk and has more coverage. It can also follow objects in several shades of orange, and the cameras have full coverage of the environment.

A. Difficulties

We ran into several difficulties when developing the robot. Because of the nature of our program, we were able to divide the work up into nodes and work asynchronously to streamline the development. However, this also made integration at the end more difficult. For example, one of the logistical difficulties we faced was that our nodes would publish conflicting messages. At the time of integration, all of our nodes either worked or were close to working properly. However, we had different understandings of which node would control the state machine, and also how frequent the state would be published. Before we fixed the problem, the robot would often miss states or go into the next state too quickly.

We also faced some difficulties with individual nodes, with the biggedt one being the `lidar_detect` node. In the `lidar_detect` node, we ran into a problem with the node unable to detect obstacles. Because our occupancy grid was formed from a collection of LIDAR readings collected from the robot patrolling the map, there were places in the occupancy grid where the walls were pixelated and did not match up perfectly to the current patrol cycle due to differences in rounding LIDAR readings. To fix this problem, we made the occupancy grid publisher also publish a binary dilation of the occupancy grid, where we blurred the walls. It took us some testing to find the perfect amount to dilate the grid: if we blurred it too much, the detector would miss intruder obstacles close to the wall, but if we didn't blur it enough the LIDAR would not have enough of a margin of error to work with.

B. Next steps:

The robot currently works well on simulation, but we did not test it on a real robot. The main next step would be to run our program on a real Husarion ROSbot in an environment like Sudikoff. We anticipate several challenges in integrating our code with a real robot. Currently, the robot mainly uses odom throughout its nodes. In real life, there would be much more odom drift, making the odom unreliable. Our localization would also have to be more robust - a brute force approach is costly and not very accurate, especially considering the fact that real data is much more noisy and prone to incorrect readings.

ACKNOWLEDGMENTS

We would like to thank Professor Alberto Quattrini Li and Mingi Jeong from the Dartmouth Computer Science Department for their invaluable help on this project and throughout the term.

APPENDIX

APPENDIX A: The Virtual Environment

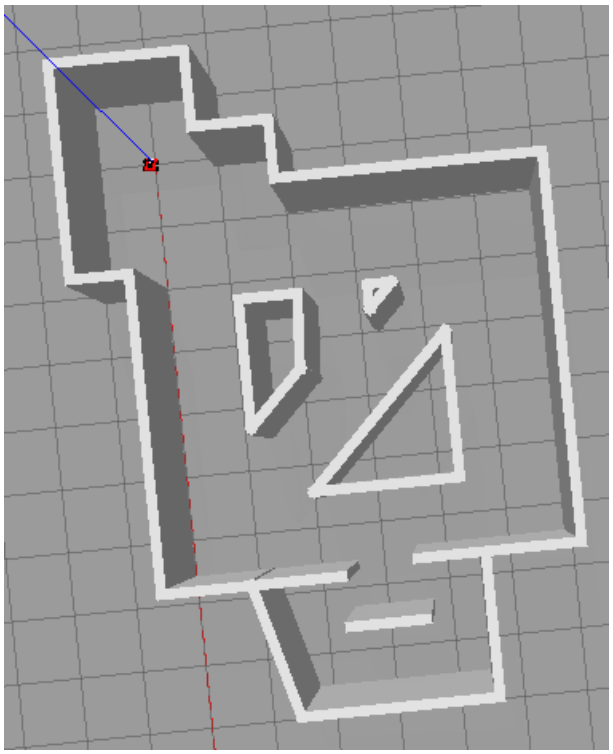

Fig. 4. The world created for testing our robot

Our robotics world was hosted within an Ubuntu Bionic Beaver virtual machine underneath the QEMU hypervisor on an Arch Linux machine. We provided it with 7 physical cores of a 5950x, and allowed it to scale between 8-32GB of RAM.

The virtual machine can be configured by following tutorials (A) and (B) and then following the instructions on the README of our Github (C). The original map was designed on paper, and produced in Gazebo by placing walls at the extremities of the map.

A. "Ubuntu Install of ROS Melodic." ROS.org. `http://wiki.ros.org/melodic/Installation/Ubuntu`.

B. "Installing and Configuring your ROS Environment." ROS.org. `http://wiki.ros.org/ROS/Tutorials/InstallingandConfiguringROSEnvironment`

C. Use the "Environment Setup (Gazebo)" section of the README file. Then use the "Modifying the World" section to copy the provided world file in the GitHub repository into the rosbot-description world files. `https://github.com/isaac-400/cs81-final/blob/main/README.md`.

APPENDIX B: The State Machine

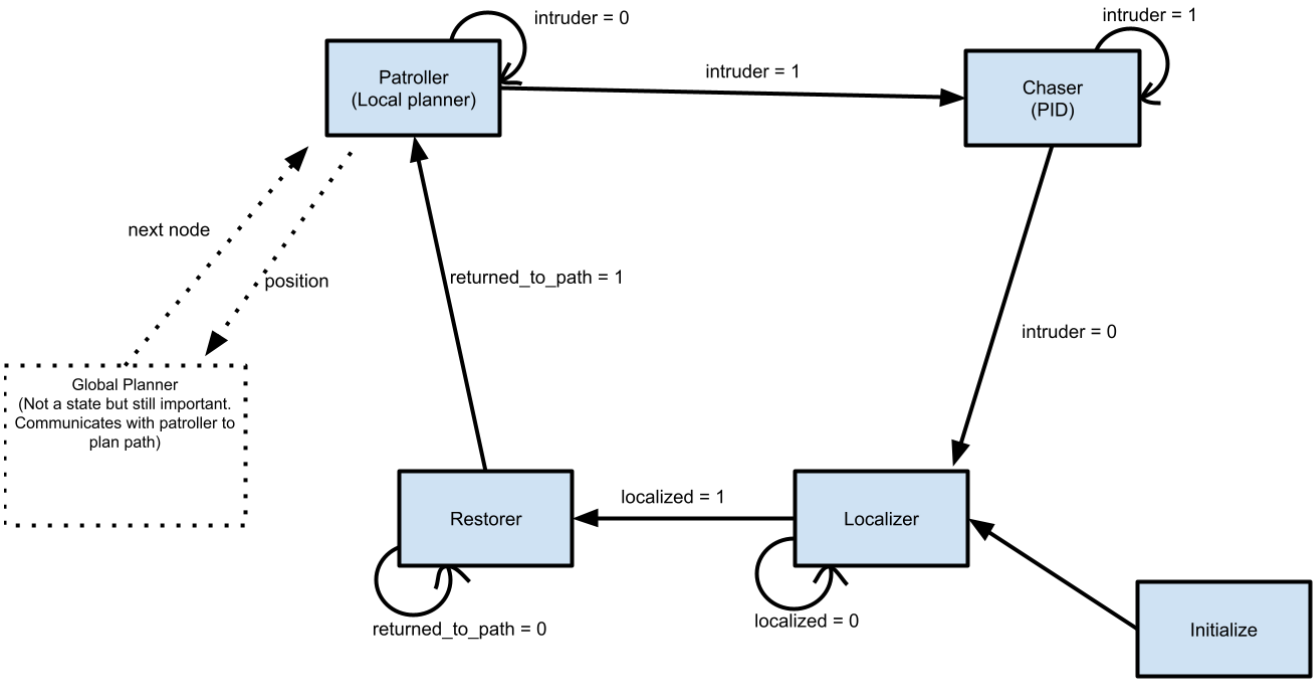

Fig. 5. The finite state machine that controls the state of the robot

The state of the robot is controlled as follows:

1) LOCALIZING. The robot localizes using LIDAR data. After localization, enter RESTORING.
2) RESTORING. The robot does breadth-first-search to compute the path to the nearest node on the Voronoi graph. It then travels to that node on the graph, and enters PATROLLING.
3) PATROLLING. The robot computes the minimal closed-loop circuit that covers all edges of the graph, and begins patrolling the graph. LIDAR object detection is performed; once an object is detected, we enter CHASING.
4) CHASING. The robot uses a PID controller and camera color detection to drive the robot toward the target. The LIDAR will return the robot to RESTORING once we get within a certain distance of the target when chasing.

REFERENCES

[1] S. Thrun, "Learning metric-topological maps for indoor mobile robot navigation," *Artificial Intelligence*, vol. 99, no. 1, pp. 21–71, 1998. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0004370297000787

[2] K. Joo, T.-K. Lee, S. Baek, and S.-Y. Oh, "Generating topological map from occupancy grid-map using virtual door detection," *IEEE Congress on Evolutionary Computation*, pp. 1–6, 2010.

[3] O. M. Mozos and W. Burgard, "Supervised learning of topological maps using semantic information extracted from range data," in *2006 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2006, pp. 2772–2777.

[4] E. Tsardoulias, A. T. Serafi, M. N. Panourgia, A. Papazoglou, and L. P. Petrou, "Construction of minimized topological graphs on occupancy grid maps based on gvd and sensor coverage information," *Journal of Intelligent & Robotic Systems*, vol. 75, pp. 457–474, 2014.

[5] H. Choset, K. Nagatani, and A. A. Rizzi, "Sensor-based planning: using a honing strategy and local map method to implement the generalized voronoi graph," in *Other Conferences*, 1998.

[6] Q. Hou, S. Zhang, S. Chen, Z. Nan, and N. Zheng, "Straight skeleton based automatic generation of hierarchical topological map in indoor environment," in *2021 IEEE International Intelligent Transportation Systems Conference (ITSC)*. IEEE, 2021, pp. 2229–2236.

[7] M. Özcan and U. Yaman, "A continuous path planning approach on voronoi diagrams for robotics and manufacturing applications," *Procedia Manufacturing*, vol. 38, pp. 1–8, 2019, 29th International Conference on Flexible Automation and Intelligent Manufacturing (FAIM 2019), June 24-28, 2019, Limerick, Ireland, Beyond Industry 4.0: Industrial Advances, Engineering Education and Intelligent Manufacturing. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2351978920300019

[8] A. CHARPENTIER, "Solving the chinese postman problem," 2018.

[9] P. Beeson, N. K. Jong, and B. Kuipers, "Towards autonomous topological place detection using the extended voronoi graph," in *Proceedings of the 2005 IEEE International Conference on Robotics and Automation*. IEEE, 2005, pp. 4373–4379.

[10] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. J. Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. J. Carey, İ. Polat, Y. Feng, E. W. Moore, J. VanderPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt, and SciPy 1.0 Contributors, "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python," *Nature Methods*, vol. 17, pp. 261–272, 2020.

[11] S. van der Walt, J. L. Schönberger, J. Nunez-Iglesias, F. Boulogne, J. D. Warner, N. Yager, E. Gouillart, T. Yu, and the scikit-image contributors, "scikit-image: image processing in Python," *PeerJ*, vol. 2, p. e453, 6 2014. [Online]. Available: https://doi.org/10.7717/peerj.453

[12] Shahbaz, "Understanding time complexity calculation for dijkstra algorithm," https://stackoverflow.com/a/26548129, 2014.