Robot Planning and Its Application

Project Report

Seyyed Arya Hassanli - 212168

February 2021

Image Transform

Calibration

The captured image from the camera is not useable directly. This happens because of the imperfection of the camera module lens. The lenses used in cameras do not have a fixed focal length among all their surface. Therefore, different parts of an image will be captured by different focal lengths. This leads to radial distortions such as pincushion and barrel distortion.

The first step to solve this issue is to find out how the camera lens affects the picture and create the camera model out of it. The camera model consists of focal length in two axes, the optical center of the lens, and five distortion coefficients.

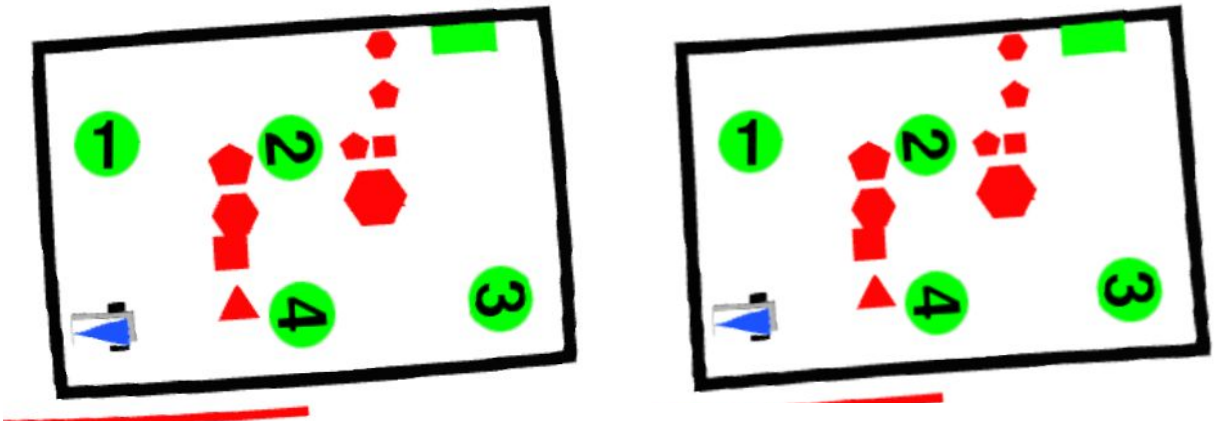
Camera Matrix: $\begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$

$$\text{Distortion}_{\text{coefficients}} = (k_1, k_2, p_1, p_2, k_3)$$

The OpenCV has a predefined function itself. It uses some samples of a chessboard image captured by the camera to calculate the camera model. The provided sample code for calibration is used in order to calculate the camera parameters. This program needs an XML file including the list of images to be used in calibration. To capture the chessboard images, it is possible to use the image listener function, which is able to produce the XML automatically to be given to the calibrator.

Undistortion

Having the camera matrix and distortion coefficients, the undistorted image could be produced by applying a transform on the input image.
The OpenCV `cv::initUndistortRectifyMap` generates the transform map from Camera Matrix and Distortion Coefficients. Then it is needed to apply the generated transform map on each frame to achieve the undistorted image. The transform application has been made by `cv::remap`.



Original Frame vs. Undistorted Frame

Extrinsic Calibration

Unwarping

Map Building

The next step toward planning the mission is to identify the map and the objects in it. There are four types of items in the arena; The robot, obstacles, victims, and the gate. Its color can identify each object.

Pre-Filter

Before any image processing, it is necessary to perform a noise reduction filter to prevent false positives in object detection. The filter used in this application is composed of three filters. The first one is a Gaussian Blur to smooth the image's sharp edges and reduce the noise. An erode, and a dilate filter will then be applied to remove more noises, fill the holes in objects, and remove the unwanted pixels close to objects.

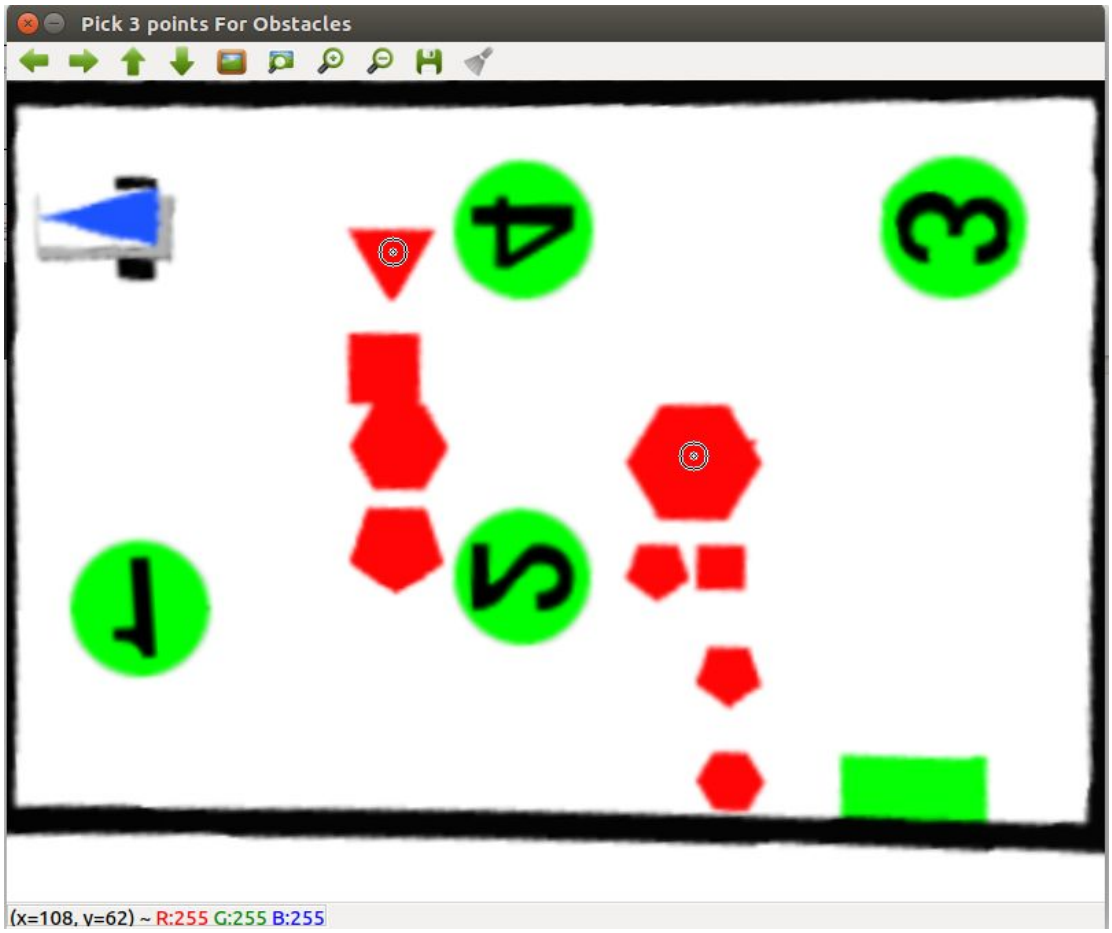
Color Masks

The first step to find objects is to filter a mask to keep the same color objects on the map. For example, to detect the obstacles' location, the mask will filter out any pixel without red color.
As far as the filter is based on the color, it would be more efficient to use HSV color space because the H value in HSV represents the color tone. So, the default BGR space of the CV should be change to HSV.

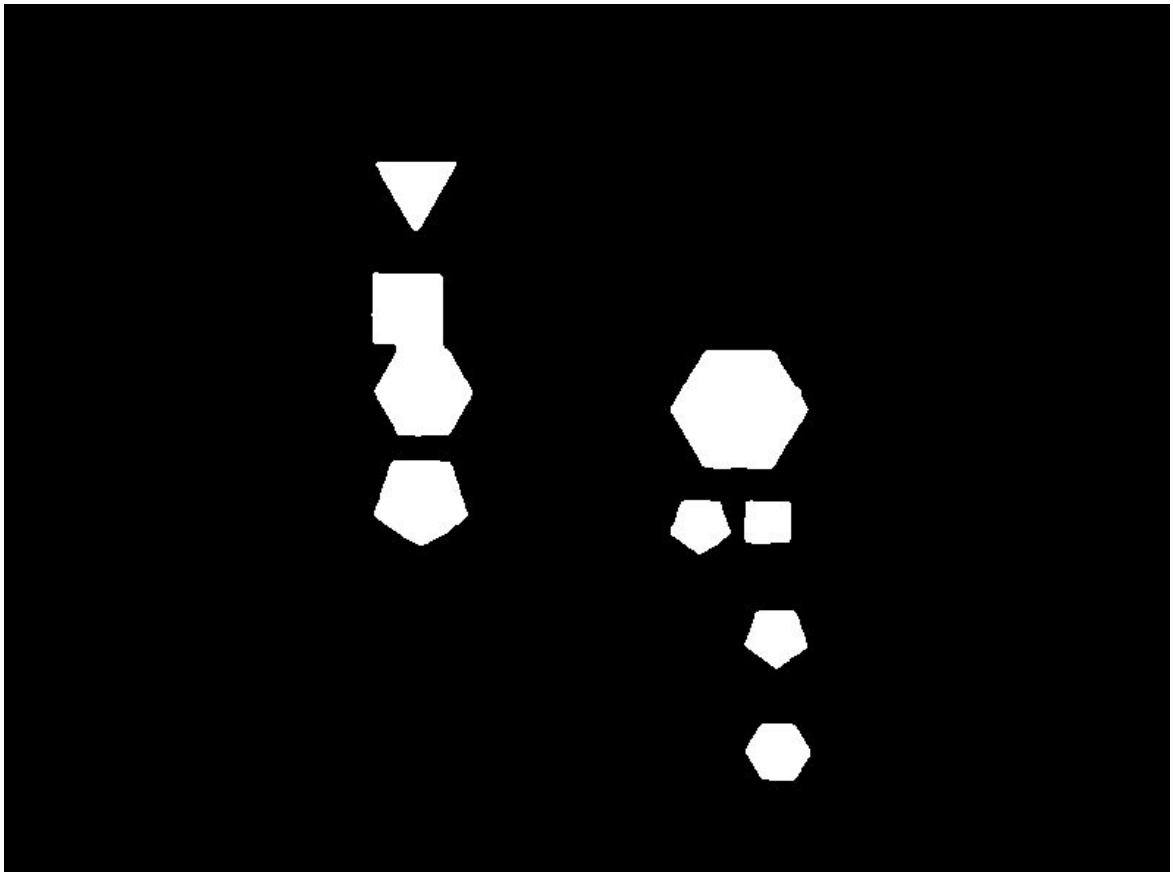
In the simulated arena, colors are not subjected to change because of environmental situations such as lighting. Therefore, it is possible to find a suitable HSV range for each object and filter the map using it. However, in the actual arena, the color tones vary by changing the lighting and even camera lighting configurations.

On the first launch, a point picker will appear. It will ask for picking three points related to each object type. It is better to pick points with different color tones. After capturing each selected point's HSV value, the program generates a mask boundary by applying a threshold. The threshold used in this project is +/- 5 for H and +/- 50 for S and V. At last, the three masks for three picked points will be added together using a bitwise OR. This way, the final mask for each object type consists of all the pixels with different color tones related to the same object.

After the first launch, the colors picked during the first launch will be saved on the config folder to be used in the subsequent launches. There is a dedicated file for each object type in which each line in the saved CSV file represents a mask with six numbers. Three numbers for HSV of the lower bound and three numbers for the HSV of the upper bound;



The Point Picker window: Two points for the Obstacle object type is picked.



Obstacles' Mask

Find Objects' Position and Shape

After creating the black and white image for each object, finding objects by finding contours in the image is possible. The contour may be useless as it has many vertices because of the picture's imperfections and noises. Therefore, an approximation Polygon of the contour will be used.
Both mentioned steps, finding contours and approximation, is handled by CV functions. `cv::findContours` and `cv::approxPolyDP` are used accordingly.

In victims' case, the additional constraint is that the approximate polygon should have many vertices as it is a circle in the actual arena. Therefore, any detected polygon with less than eight vertices is ignored. The other constraint is related to the gate. The gate should have precisely four vertices.

OCR

Each victim on the map has a number on it. The number should be extracted to be used later for mission planning.

In the beginning, the victim is extracted from the frame by cropping the victim's bounding box. Then two steps of gaussian blur, erode and dilate, will be applied to remove noises from the image of the victim's number and make it smoother.

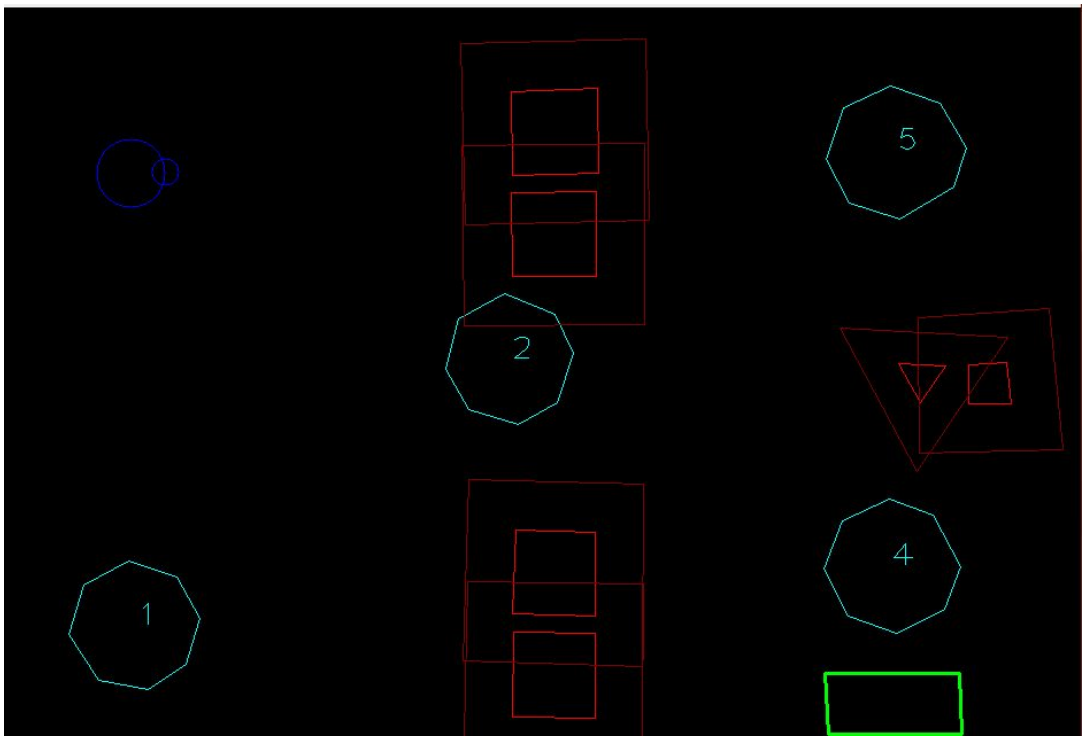
The OCR library used in this project is the tesseract-ocr. One of the challenges of using tesseract was that it did not have a perfect configuration to auto-detect a single character's orientation. However, it could output confidence measure.
To solve this challenge, the victim's image is given to the ocr library with every possible flip and rotation, in a total of eight states. The extracted number with the most confidence will be used for further steps.

Recognized digit: 2, Confidence: 89

Planning

Create Margin for Obstacles

The easiest way to plan the robot's motion in the map without touching the obstacles is to create a margin for obstacles and assume that the robot is a single point. In this way, the collision check is much easier.
As all of the obstacles are convex polygons, the method used to make a margin for barriers is to move the vertices by a fixed amount from the center.



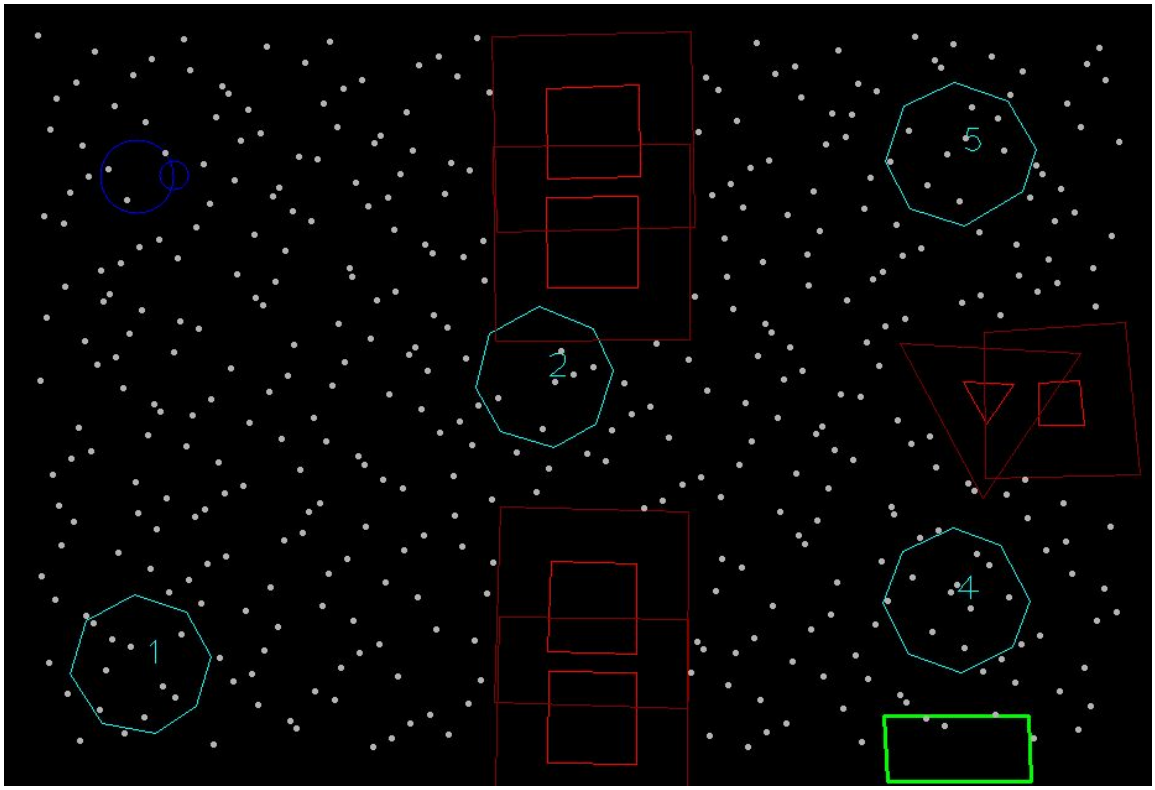
The obstacles and the added margin

Create Random Points

The first step of converting the map to a graph is to define the graph nodes. The method that used in this project is generating random points using Halton's method. Four hundred ninety-nine points are generated in the map using the Halton method considering a margin for borders.

Remove Redundant Points

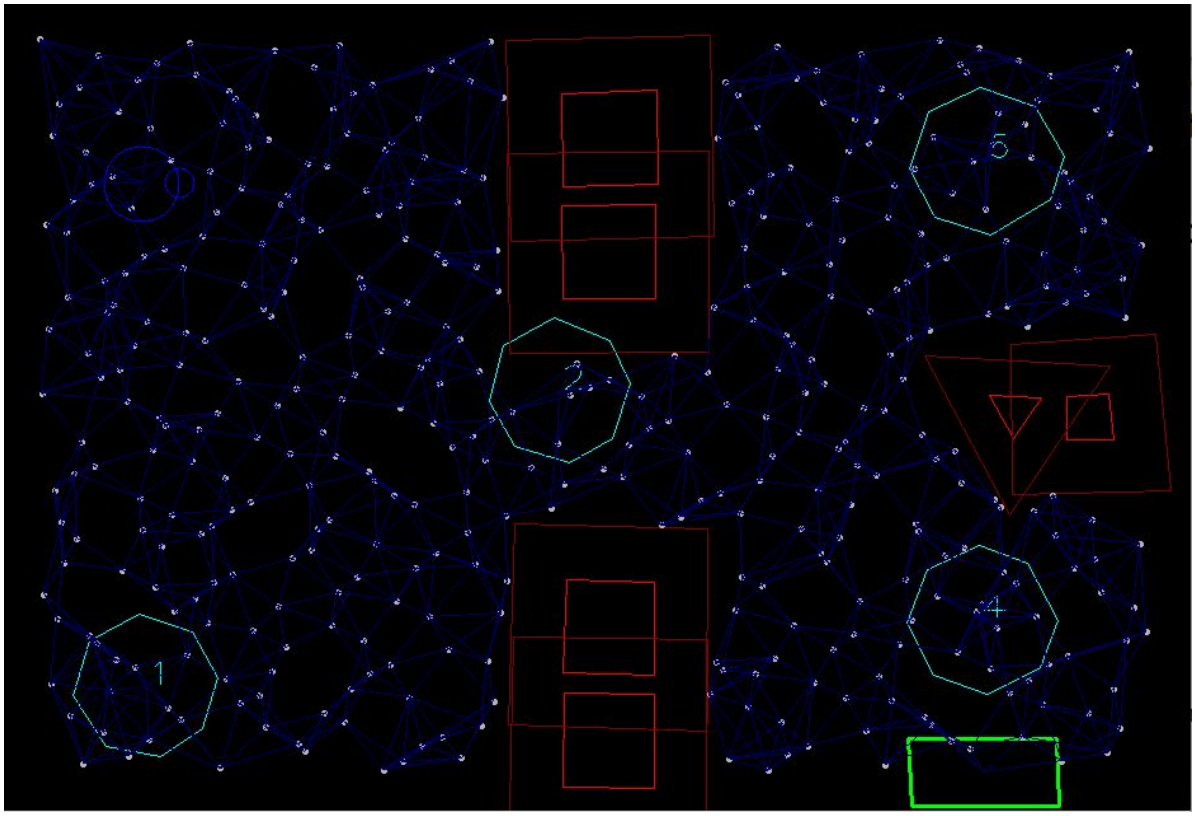
Some of the generated Halton points are either inside of the obstacles or inside of the margined obstacles. These points are useless because they cannot be visited. In this step, points that are inside of the margined obstacles are identified using cv::pointPolygonTest and removed from the points list.



Creating the Graph

The graph is defined as a list of Node objects. Each Node holds the x and y position, list of neighbors, and some other data to be used in other processes. First of all, one node will be added to the graph for each eligible Halton point. Then other object nodes are added; one for the robot, one for the gate, and one node for each victim.

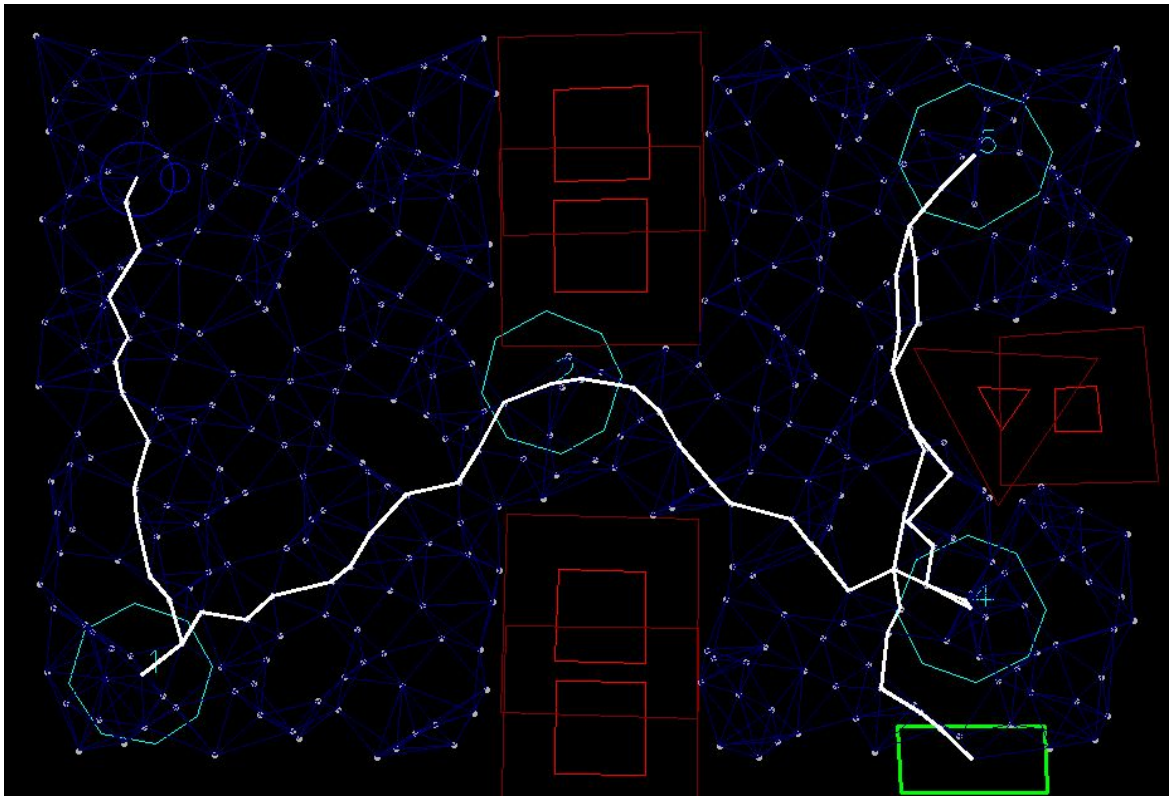
The next step is to create the graph's edges. Edges are made between the nodes that are defined as neighbors. To find the nearest neighbors, the map is divided into ten rows and ten columns, a hundred sections. It is assumed that each node's nearest neighbors are either in the closest areas around the node's section or its division. Therefore, these nodes will be checked, and their distance to the selected node will be measured. The maximum of K nearest neighbors is considered the node's neighbors, and the edges are created accordingly.



Find Route

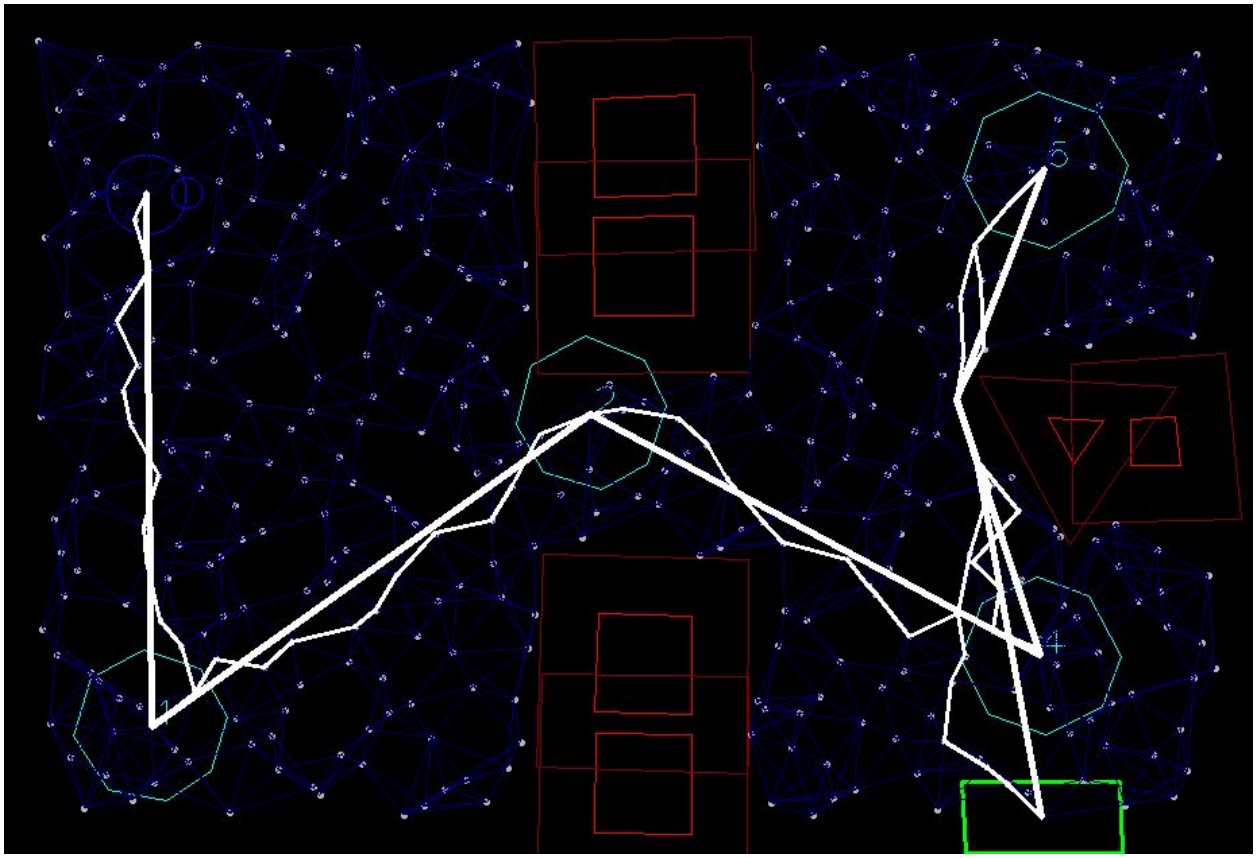
Now that the graph is created, the next step is to find a short path between the desired nodes. For the first mission, the order of the nodes to visit is apparent. The path starts from the robot node, goes through the victims in their order, and ends in the gate node. These nodes will be pushed to the checkpoints list. Then, the program starts to find a path between the nodes in the checkpoints list.

The shortest path algorithm that is used in this project is Dijkstra. Dijkstra provides the shortest path starting from a source node. Based on the method of implementation, the complexity order could be $(|V|+|E|)\log |V|$ or $|V|^2$. It is also possible for better performance to terminate the calculations when the Dijkstra reaches the target node.



Smooth the Path

The routes that are found by Dijkstra can be made smoother by taking shortcuts. A recursive function gets a route and tries to find out if it is possible to go directly from the source to the target without touching any obstacles. If it is possible, it returns the new route directly from the start to the target. But if not, it repeats the check for a possible shortcut between the start and a middle node and between the middle node and the target.



The Actual Motion Plan

Finally, the actual motion plan is generated by calculating a Dubins curve between each starting point and endpoint. The curvature used for this calculation is 40.