Robotic Cluedo

Report

Mohamed El Mehdi Amine
sc17mema

Introduction

This report is an overview of the work we achieved to implement Robotic Cluedo. The main aim of this report is to present our solution by describing our design, justifying the decisions we took, showing results of the testing, and analyzing our design through its strengths and limitations. After that, the report reflects on my and my team's assessment.

Design

Our approach towards designing a solution was to look at Robotic Cluedo in terms of subtasks that a TutleBot should perform in order to recognize the images in the arena. It was clear that the design should answer two main problems, a navigation problem and a computer vision problem.

The navigation problem consists first of enabling the TurtleBot to move to some given coordinates of the center of the map. After that, the robot should be able to navigate the arena in such a way that would enable it to search for the images.

As the ROS Software provides path planning and map localization, enabling the TurtleBot to go to the center of the map was equivalent to writing a script that takes the coordinates of the navigation goal and moves the robot to that point. Making sure that the TurtleBot can navigate the arena was more complicated to approach.

The arena could be considered as a maze, hence one way to solve the navigation problem is to rely on a maze solving algorithm. For example, we could do some image processing on the map of the arena to detect the walls, set a number of points along those walls, then have the robot go along the walls and stop at each point, spin around itself, and see if it detects an AR marker. In case it detects an AR marker, move to a certain distance from it and take a snapshot of the image. This approach is similar to a Bug Algorithm in which the robot wouldn't bump into obstacles but rather plan a traversal of the walls detected from the map. Except that in this case we don't know the navigation goal, so we can't make decisions on when the robot should stop going along the borders of one obstacle and move towards another obstacle.

A simpler approach that allowed us to speed the design process is to generate random points in the map and have the robot move from one point to the other while stopping at each to see if it can detect an AR marker. This is not efficient because the completeness of this algorithm relied on eternally generating points as long as no images have been detected. It's also not efficient because the robot was following the points in the order of their generation. Clearly this could be enhanced for more performance.

By calculating the distances between the generated points, a distance matrix can be produced. Then based on the distances stored in the distance matrix, we can order the points in such a way that the robot always goes to the next nearest point. This is more efficient as it saves time and makes the robot move in an ordered way as opposed to the arbitrary movement we got at first. This is a little bit similar to the first phase of probabilistic roadmap planning. But different in so far as we don't consider collision points while connecting the generated points. Because the TurtleBot is already able to plan paths without bumping into collisions. Another difference is that we don't know our end goal, so we are not constructing a graph towards it. Rather generating a shortest-path graph between the generated points.

Of course the randomly generated points should give x and y coordinates within the boundaries of the map. We were first relying on setting a range for the generated points by using Rviz. We would check the left most, right most, top, and bottom points from Rviz. Then we would generate points within those boundaries. But our design had to automate this procedure because such a maneuver is not allowed during the demo.

Luckily, the wiki.ros.org website provided a script for cropping the image of the map from a yam file, then returning information such as the resolution, the origin, and the boundaries of the map. We took advantage of this script to get these values and then compute the real boundaries of the map. The real boundaries of the map being the origin added to the product of the boundary times the resolution. Small changes were brought to the cropping script (mapping.py) such as transforming the main function to take a yaml file as an argument and return the resolution, origin, and boundaries.

At each point, the TurtleBot spins to scan its surrounding in order to detect an AR marker. In case an AR marker is detected, its location is recorded in terms of its coordinates and the orientation that the robot should take in order to face the image. A check is made to ensure that the robot doesn't go twice to the same marker. The check is performed based on the coordinates of the already detected AR marker if there is any. Once the check is performed, the robot can position itself within a set distance from the image. This is necessary to ensure that we can take a good snapshot of the image detected.

We did not consider any alternative to AR detection as the project guideline encouraged us to use it and that was what we learned from the lab sessions.

Once the snapshot is taken, the computer vision part of the design starts. We approached this part by subdividing it into weapons recognition and character recognition. We noticed that the weapons share a similar background color and it would make sense for an algorithm to first distinguish whether the image is of a weapon or a character before recognizing which weapon or character it is. By drawing contours around the characters and performing color detection on the contours of the background, it is possible to determine which character is in the image. The same cannot be achieved with the images of the weapons as they share similar background color. Hence thinking in terms of color isolation and detection was solving only part of the problem.

The design we opted for is performing feature matching and treating weapon images and character images similarly. Feature matching is performed on the snapshot taken and uses the images of the project as a training set. Images are processed in 3 color channel and the algorithm proceeds by finding the keypoints and descriptors for the training set as well as the snapshot. Brute-force matching is then used to determine feature matches between the snapshot and each image in the training set. These matches are then sorted from highest number of matches to lowest. The algorithm then determines the image with the highest number of matches among the training set as the snapshot image. The keypoints of this image are used with those of the snapshot to make a homography matrix. The homography matrix is useful because it records the transformation from each keypoint in the snapshot to the keypoint in the image with the highest number of matches. After this, the height and width of the snapshot are used to find its corners. Then the homography matrix together with the corners of the snapshot, are used to apply a perspective transform on the corners of the snapshot corresponding to the homography matrix. A polygon is then drawn around the matching image by using the new corners computed from the perspective transform.

Testing and Results

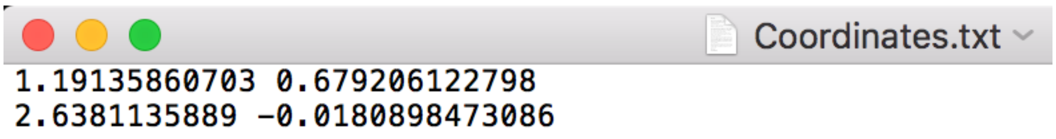
The main testing was performed in the robot lab, but some simulation testing was performed prior to that for the map cropping and the feature matching scripts.

This video shows the robot navigating the map and stopping at each point to spin and check for AR markers. After that it positions itself in front of the AR marker, takes a snapshot, processes it and outputs a text file with the coordinates of the AR marker along with processed images after feature matching.

Video link of the TurtleBot performing:

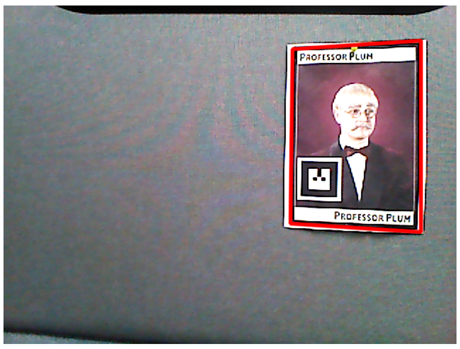
https://leeds365-my.sharepoint.com/personal/el15kd_leeds_ac_uk/Documents/Robotic%20Cluedo%20Turtlebot%20Test%202.mp4?slrid=de27339e-b0fc-4000-caac-797e066244e0

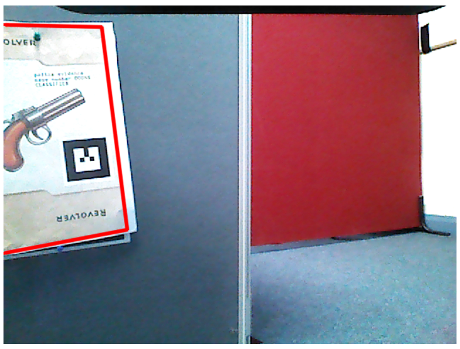
Screenshots of the text file with the coordinates of the detected AR markers:



```
1.19135860703 0.679206122798
2.6381135889 -0.0180898473086
```

Feature matching tests on snapshots with different quality:





Strengths and Weaknesses of the Design

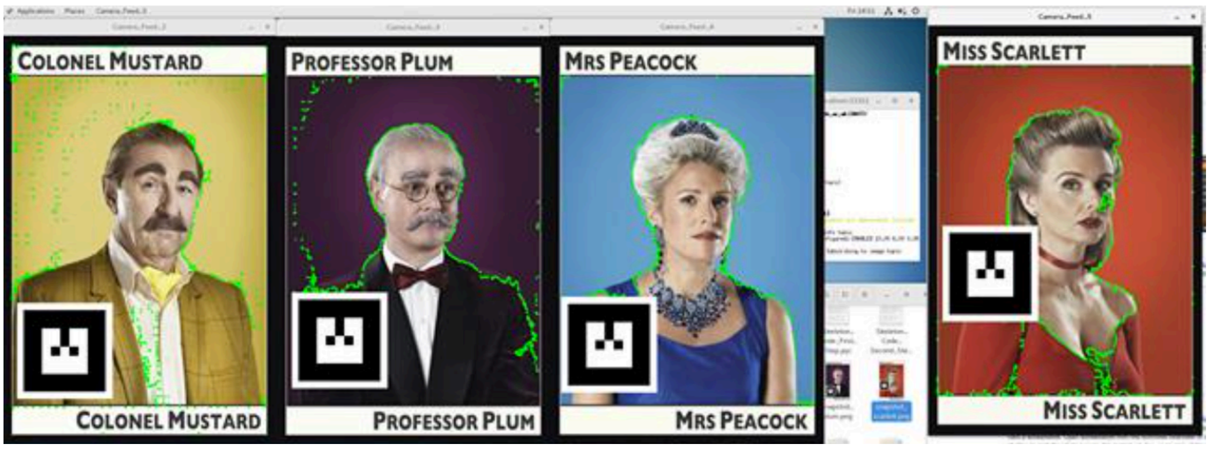
Relying on randomly generated points relies on the number of iteration we perform. The more iterations, the more likely we are to visiting different points in the arena and finding an image. Striving for better performance, special care has been given to optimize the implementation of the graph construction that determines the the journey of the robot. Loops have been avoided and reduced to one. The distance matrix is efficiently implemented using the SciPy's method distance.cdist(). Also, being able to crop the image of the map and to use it to get the boundaries of the map allows to generate points within the needed range. The cropping script is provided by ROS which is the first go to resource for a TurtleBot project. This method applies to all kinds of maps but the cropping script is severely affected by any noise around the edges of the map. Sometimes mapping doesn't produce clear borders of the map and generates some noise that extends beyond those borders. This makes the map larger, which means the cropped image is also larger. Thus the boundaries are not accurate and the generated points could fall into the noise region that is actually not part of the real map. A work around this was to always select a few points among the generated ones so that the the farthest points are never visited. So less likelihood for the robot to have to deal with points beyond the map's region.

During our testing, the feature matching algorithm implemented never produced a wrong output. It's definitely the greatest strength of our design as it was reliable even when the snapshots taken were of mediocre quality. The script makes good use of the cv2 library to perform feature matching that works on images whether they are of weapons or characters. It could certainly reduce its computations by determining at first if the image is of a weapon or a character. This way it could select only the necessary images from the training set of images. The fact that it relies on a training set could mean that the performance is lowered with a greater number of images to search from.

Self and Team Assessment

I contributed to the project by implementing the path planning algorithm and finding a solution to automating the procedure of setting the boundaries of the map. I found the cropping script in the ROS wiki and examined it and made necessary changes before importing it in our main script go_to_specific_point_on_map2.py. I used this script with the help of Tom to generate the real boundaries by calculating them from the cropped image boundaries times the resolution plus the origin. My code then generates random points within the boundaries computed and establishes a distance matrix from which the path that the robot will go through is sorted in terms of which is the next closest point.

Konstantin worked on contour finding and contributed by implementing a script that draws contours around the characters. This solution was later on abandoned as we opted for feature matching. It was fallible as the drawn contours were not accurate.



A good example to illustrate this is the image of Mustard. Image thresholding wouldn't be efficient with the kind of contours we got from this approach.

Marcus worked on the feature matching script and implemented all of it on his own. I admire him for his work.

Tom made use of the path I generated to make the robot navigate and perform AR detection. Then he relied on the feature matching script after taking the snapshot of the images.

Our group worked independently on the different parts of our design and we did not need to physically meet as we often talked on Facebook. At first I had many urgent assignments to submit and my group was quick and efficient. By the time I was done with my assignments my group had done considerable progress. I caught up to the progress made and examined each of the scripts already written, then was able to enhance the navigation code and make use of the map cropping code (mapping.py) after bringing some changes to it. I was also involved in the testing phase and added some code to output the name of the image along with its coordinates in a text file. This addition was later on abandoned as the project guideline specifies that the text file should contain the coordinates and nothing more was mentioned.

My contribution to the testing was focused on making sure that the robot goes to the center of the map and travels from one point to another while respecting the generated path. Tom and I also played with testing different spinning speeds and see the effect it had on making the robot faster without affecting the quality of the snapshots, because the robot spins to face the image before taking the snapshot and we thought it was too slow.

My contribution to the wellbeing of the team was during the some lighthearted moments of our work. I also did my best to answer any questions or any need expressed by one of my teammates. Tom forgot his Facebook account logged in the TurtleBot laptop before leaving once. I scared him by sending him a screenshot of his opened Facebook page. Then I innocently signed off without doing any misdeed. I came to regret that very quickly as I also did the same mistake of forgetting to log off my Facebook in the same laptop. He wasn't as forgiving as I was. He sent random messages from my Facebook. We had both productive and not so productive kinds of fun.