Mechanical & Industrial Engineering
UNIVERSITY OF TORONTO

CONTEST 2 REPORT

UNIVERSITY OF TORONTO

MIE443 - MECHATRONICS SYSTEMS: DESIGN AND INTEGRATION

Finding Objects of Interest in an Environment

Authors:

Project Group 18

Farhan Wadia — 1003012606

Henry Cueva Barnuevo — 1003585122

Yilin Huang — 1003145232

Professors:

Prof. Goldie Nejat

Dr. Meysam Effati

Head Course TA:

Pooya Poolad

March 25, 2021

Table of Contents

1.0 Problem Definition

This problem is the second in a series of three contests related to the control and navigating capabilities of autonomous robots. For this second problem, a simulated TurtleBot will be required to autonomously navigate an environment to find and identify ten known objects and return to the starting position within 8 minutes. This navigation will be based on sensory information from the Microsoft RGBD sensor, bumpers, and odometry, which are described in more detail in Section 4.0, and the image recognition will be handled by the Microsoft RGBD sensors using OpenCV to perform SURF feature detection, which is described in more detail in Section 5.1.2.

2.0 Functional Requirements

The objectives and constraints for this contest are discussed in the subsections below.

2.1 Objectives

In accordance with the contest requirements stipulated in Section 2.2, the following objectives have been identified:

1. The TurtleBot should minimize the amount of time travelling to the 10 objects (i.e. must optimize the navigation path)
2. The TurtleBot should maximize the accuracy of the images identified and minimize the amount of time spent identifying images.

2.2 Constraints

The problem itself has several constraints, including:

1. The environment that the TurtleBot is exploring and mapping is contained in a 6x6 m^2 2-dimensional simulated environment with static objects
2. The TurtleBot must navigate the environment, find and identify the 10 objects, and return to the starting position in a maximum of 8 mins. It must also stop moving once it is done scanning or when the 8 minute period is over (whichever occurs sooner)
3. The objects that the TurtleBot are identifying are faces on boxes of size 50 x 32 x 40 cm^3 (l x w x h). The image will be located on the longer face of the box
4. A dataset of all possible images will be provided to the team before the contest day
5. The information provided to the team on the contest day will include:
    a. A 2D map of the contest environment (without the objects)

   b. Test locations for the 10 objects (given as a location (*x,y*) and orientation as a rotation about the *z* axis)

6. Must use OpenCV to perform SURF feature detection for image recognition
7. At the end of the contest, the TurtleBot must output a file with:
   a. All tags it has found (output tag ID)
   b. Order of the tags found
   c. Coordinate of the tag
   d. Whether the image is a duplicate or not

3.0 Competition Strategy

The competition strategy is subdivided into a strategy for path planning and navigation, and for image recognition.

3.1 Path Planning and Navigation Strategy

The contest requirements, as stipulated in Section 2.2, are to identify boxes at known locations in an environment, return to the starting location after identifying all the boxes, and to optimize the path. This means that navigation for this contest can be reduced to solving and implementing a solution to an instance of the Travelling Salesman Problem (TSP).

To model the contest as a TSP, we abstract it as an undirected graph. The 10 boxes are considered vertices of the graph, and we form a complete graph by creating edges that connect each vertex to all other vertices in the graph. For two vertices u, v, we can define $w(u, v)$ to be the weight of the edge connecting those two vertices, and we define each weight to be the Euclidean distance between the two vertices that the edge connects. The TSP is then to find a Hamiltonian cycle of minimal weight in the graph (i.e. a simple cycle of all the vertices in the graph where the sum of edge weights in the cycle is minimized).

TSP is a well-known NP-Hard problem, which means that no polynomial-time algorithm is currently known that can provide an optimal solution; in other words, the only way to find the optimal cycle is to enumerate through every possible cycle. This would have a time complexity of $O(V!)$, where $V$is the number of vertices in the graph. We thus considered two approximation algorithms to try and solve the TSP in polynomial time with nearly optimal solutions.

The first algorithm we considered works by using Prim's algorithm to find a minimum spanning tree (MST) of the graph in $O(ElogV)$time (if implemented using a binary heap), and then doing a pre-order walk of the MST in $\Theta(V)$time. Note that E is the number of edges in the graph, and being a complete graph, $E = \binom{V}{2} = \frac{V(V-1)}{2}$. Thus, the runtime of this algorithm is $O(V^2 logV)$, and can be shown to always provide a cycle whose weight is no more than twice that of the optimal cycle [1]. As a second alternative, we considered Christofides' algorithm, which runs in$O(V^3)$time, and is guaranteed to find a cycle of weight no more than 1.5 times the weight of the optimal cycle [2]. Both of these approximation algorithms are valid solutions to our instance of the TSP because our graph edge weights are Euclidean distances satisfying the triangle inequality, which is a necessary condition for being able to use these algorithms.

Although we initially considered using approximation algorithms for the TSP, given that the number of boxes in the problem will always be 10, we decided to use "brute force" and check through all path possibilities in order to get the optimal strategy. As shown in Table 1, running a brute force solution for a 10 vertex TSP returns in under a second, but the runtime gets significantly larger as more vertices are added because of the $O(V!)$ asymptotic runtime. In our own implementations of the brute force strategy, we also achieved similar runtimes. Thus, given the relatively low execution time and guarantee that the contest environment would have no more than 10 boxes, we decided to keep the brute force strategy and did not pursue trying to implement approximation algorithms.

Number of Vertices	**10**	**11**	**12**	**13**	**14**
Average Execution Time (s)	0.242	2.438	27.845	348.849	> 900

Table 1: Average execution time for 3 trials of brute force TSP solutions for different graph sizes, implemented in C++ on a MacBook Pro with 2.4 GHz Quad-Core Intel Core i5 processor [3].

Although theoretically our brute force strategy should guarantee an optimal solution to the problem, there are a few reasons why we cannot guarantee that our solution is exactly optimal. In the contest, we may start at any location and must end back at that same location, but our solution only finds the optimal cycle between the boxes; it did not consider the starting location as a vertex in the graph. To save on computational time, rather than implementing an 11 vertex graph by including the start point as a vertex, we only implement the 10 vertex graph, start the tour at the box closest to the starting point, and directly return to the starting point after visiting the last box of the path. Our strategy therefore could yield paths that are sub-optimal since we did not consider the starting position as a vertex in the graph. However, we feel this is a reasonable compromise to save on computational time; given the effects of running in a simulated environment, our execution times are slightly longer than those listed in Table 1.

In addition, our graph edge weights using the Euclidean distance between two boxes is oversimplified; using this does not account for obstacles that may lie in between a straight path between two boxes. The distance of the path between two boxes only equals the Euclidean distance if there are no other obstacles or boxes in between; when there are obstacles, the distances can easily become longer. Nevertheless, our strategy meets the contest requirements since it tries to optimize the navigation path, but may not provide the optimal path for the reasons outlined above.

3.2 Image Recognition Strategy

As described in Section 2.2, SURF feature detection to determine the tags corresponding to all the boxes in the environment is required. Additionally, it is required to identify if the images are duplicated or not. In order to achieve this, our first decision was to only activate the image recognition after the robot finished navigating to each location. This saves computing power as the robot is not processing images continuously, but only as needed. Also, it assures that the robot utilizes the correct input, as it captures a single image after it reaches the desired box and stops moving, increasing the accuracy.

SURF was applied iteratively, following the instructions in Tutorial 4 [4], as well as information found in OpenCV feature matching tutorials [5][6]. For each tag, a FLANN matcher was chosen to compare with the box since it only considers the nearest neighbours, making it fast and efficient. We then applied three methods to determine whether that image tag is the best match.

First, the matches were filtered using Lowe's Ratio Test [6]. This method compares each "acceptable" match's distance with the distance of the second-best one. If it is below a certain ratio, it passes the test and is stored as a good match. This method is chosen because it uses a ratio instead of a definite value, making its application agnostic, and in turn easier to apply. Additionally, it is widely utilized and its performance has been tested. In our case, the team chose 0.75, as it is defined to be the optimal ratio [6].

Second, the team uses homography to map the box image capture to the template tag, define a contour using its corners, and find the area of this contour [5]. We use this area to determine if a match was likely to be valid or not by using upper and lower boundaries. If the area falls within the acceptable range, it is assigned a weight of 1. Else, it is assigned a weight of 0. At the final step, this weight is multiplied by the total number of matches for that tag. This helps us discard cases where we could have a high number of matches, but the area is much smaller or larger than expected, indicating a possible false match.

Third, each of the matches that passed Lowe's Ratio Test is checked to see if it is within the area contour defined through homography. If so, it is accepted; otherwise, it is discarded. This strategy also helps us eliminate false matches, increasing the accuracy of the algorithm

Finally, the three-step process described above is repeated for each image template. A variable with the current number of best matches is initialized, with the minimum number of matches for an image to not be classified as blank. For each iteration, if the number of matches is higher than the current number of best matches, then it is replaced, along with the id of the current template. This id is then stored in a list, which is used to compare and find duplicates, and is then written to our output file.

This strategy uses various tests and filters to find the best match, and in our testing, successfully identifies each box in an efficient and effective manner.

4.0 Sensor Design

The simulated TurtleBot comes equipped with multiple sensors, including several of which are built into the Kobuki base. A layout of where these sensors are located is shown in Figure 2. For this contest, the bumpers, odometry, depth sensor (part of the RGBD sensor), and IMU are used. Each sensor and how it was used to meet the contest requirements are described in more detail in the following subsections.

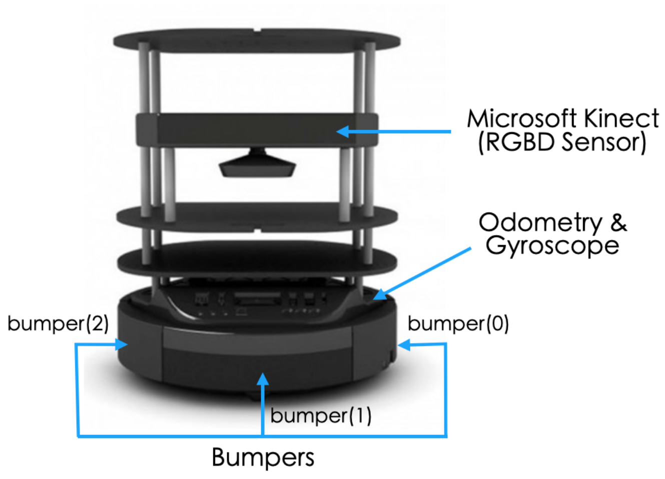

Figure 2: Placement of sensors on the simulated TurtleBot (Front view).

4.1 Microsoft Kinect RGBD Sensor

The Microsoft Kinect RGBD sensor is composed of two sensors, the RGB sensor and the depth sensor, both of which are used for this contest.

The distance sensor projects a speckle pattern of IR beams onto objects in the environment, which are then reflected and detected by the IR depth sensor. The distortion of the original pattern from this reflected pattern determines the depth information.

The RGB sensor can detect red, green, and blue components, as well as body-type and facial features. Feature recognition is utilized by the SURF detection algorithm, which is described in detail in Section 5.1.2.

4.2 Odometry

The odometry sensor is used to estimate the TurtleBot's position and orientation relative to its starting location. From this sensor, we use `posX`, `posY`, and `yaw` to represent the horizontal position, vertical position, and angular orientation of the TurtleBot relative to its starting position.

The path planning part initially determines the coordinates of the vertices the TurtleBot will visit as (x, y, orientation). The odometry of the TurtleBot is then used to match these values to ensure it has arrived at the desired location. This is explained further in Section 5.2.1.

4.3 Bumpers

There are three bumpers on the TurtleBot's Kobuki base located on the left, center and right sides of it, as shown in Figure 2. The bumpers are used when there are obstacles that the RGB sensor cannot see, such as the brick wall in the practice world simulations. The bumpers are initially set as released, with a state of 0, while a pressed bumper returns a state of 1. Unlike in Contest 1, where the TurtleBot is navigating an unknown environment, in Contest 2, the map of the environment is given beforehand and a path planning algorithm ensures the robot will not collide with any obstacles (explained further in Section 5.1.1), so the need and use of the bumpers in this contest is limited.

5.0 Controller Design

Section 5.1 discusses the high level control architectures implemented in our program, and Section 5.2 discusses the lower level control architectures of the program.

5.1 High-Level Control Architecture

Our program's high-level control architecture can be further subdivided into how we implemented our path planning and image recognition strategies, which are discussed in Sections 5.1.1 and 5.1.2 respectively.

5.1.1 Path Planning

As discussed in Section 3.1, our team's path planning strategy is to model the box locations as vertices, and calculate the Euclidean distances between any two boxes as weighted edges of an undirected graph. We then treat this graph as an input to solving the TSP using a brute force approach, and decide to navigate to the boxes in the sequence that gets returned after solving the TSP. The following sections provide implementation details for how this was achieved.

5.1.1.1 Defining graph vertices

First, we take the box coordinates given by `boxes.coords[i][j]`, where `i` is a value from 0 to 9 representing indices of the 10 boxes, and `j` is a number from 0 to 2 representing the x position, y position, and angle respectively, and use it to define a corresponding `nav_coords[i][j]`. Since the coordinate system for the boxes is flush with their front faces, we cannot navigate directly to these points because it would cause the TurtleBot to hit the box, and because the TurtleBot would be too close for image recognition of the face to work well. Therefore, we define `nav_coords[i][j]` as the coordinate to try navigating to if we want to visit the box at `boxes.coords[i][j]`, and it is these `nav_coords[i][j]` that form the vertices of the graph data structure that we form.

Each value of `nav_coords[i][j]` is filled in by the **`fillNavCoords`** function of our program. Using a parameter (`offset`) that is also passed to the function, we use trigonometry to calculate a position and orientation that directly faces the centre of the box face at a distance of `offset` meters away. When **`fillNavCoords`** is called, we use `offset=0.4`.

In determining the orientation for each coordinate to navigate to (i.e. each `nav_coords[i][2]`), since we want to directly face the box, we need to add or subtract 180° to `boxes.coords[i][2]`, and the decision for whether to add or subtract is based on ensuring that the result is within a range of [-π, π], because that is the defined range of orientations that the TurtleBot uses. To do this, we use a helper function called **`minus2Pi`**; essentially, this function helps us convert intermediate workings in a range of [0, 2π] back to the required range of [-π, π].

5.1.1.2 Defining graph edges

The most common ways to represent a set of graph edges are as an adjacency list or adjacency matrix. In an adjacency list representation, we maintain *V* lists (one for each vertex of the graph), where each list contains only the edges and corresponding weights to vertices adjacent to the given vertex *v*. On the other hand, an adjacency matrix is a $V \times V$ array where each entry (i, j) of the array corresponds to the edge weight between vertex i and vertex j. Adjacency lists have the advantage of requiring asymptotically less memory than adjacency matrices, but with the disadvantage of needing to conduct a search to determine if any two vertices share an edge. With an adjacency matrix, however, this can be done in constant time by directly checking the corresponding entry in the matrix [1].

Consequently, we decided to use an adjacency matrix representation for the graph edges because of it being easier to implement (no need to have a searching algorithm to find an edge weight between two vertices), and because edge weights can be accessed quicker, allowing us to reduce time in solving the TSP. With a 10 vertex graph, the tradeoff of using more memory compared to an adjacency list and the redundancy of saving the same weight for edge [i, j] as [j, i] is insignificant.

Our implementation of calculating the adjacency matrix `adjMat` can be found in the **`fillAdjacencyMatrix`** function. It uses **`dist`** as a helper function to calculate the Euclidean distance between points (`x1 = nav_coords[i][0]`, `y1 = nav_coords[i][1]`) and (`x2 = nav_coords[j][0]`, `y2 = nav_coords[j][1]`) to fill the matrix.

5.1.1.3 Solving the TSP

Now that the data can be considered as a graph whose vertices are stored in `nav_coords` and edges are stored in `adjMat`, we can implement the brute force solution to the TSP. Our implementation is largely similar to the implementation by Tripathy, and can be found in the **`bruteForceTSP`** function [7]. Our modifications to his implementation were to allow for floating point edge weights in the adjacency matrix rather than integers, and to also modify a vector `TSPTour`, passed in as a pointer, to hold the path of vertices to visit for the optimal, lowest weight cycle.

Before beginning, we use the **`findClosestBoxAtStart`** function to find the index of the box closest to the initial position, and tell **`bruteForceTSP`** to use this vertex as the initial vertex of the path.

The **`bruteForceTSP`** function essentially works by initializing the length of the shortest cycle to infinity, and then looping through all 10! possible paths. Any time a cycle of lower weight is discovered compared to what was previously considered the lowest weight cycle, the cycle and corresponding weight are saved. After iterating through all permutations, the result is the value of the lowest weight cycle and corresponding tour, which we always set to start at the vertex closest to the TurtleBot as discussed in the preceding paragraph and Section 3.1.

5.1.2 Image Recognition

The image recognition algorithm is implemented in order to match each box image with its corresponding image template from our database. This is accomplished through the OpenCV library, taking advantage of several built-in functions. The matching process is performed through three functions: **`imageCallback`**, **`matchToTemplate`**, and **`getTemplateID`**. The first function is provided by the contest code, and uses the `"camera/rgb/image_raw"` service to get a live video feed of the RGB camera in the Kinect sensor. When called, it captures the current frame of this stream. Next, the **`matchToTemplate`** function follows the steps outlined in Tutorial 4 [4], as well as the "Features 2D + Homography to find a known object" OpenCV tutorial [5], in order to match the image captured with one of the image templates. Finally, the **`getTemplateID`** function performs an iteration of **`matchToTemplate`** through all the image templates, and writes the best one in the contest .csv file.

The matching process is as follows. Once `Navigation::moveToGoal` is successfully called, and the TurtleBot reaches the desired location, the function **`getTemplateID`** is called for that box, using as the argument our set of image templates. Inside this function, **`matchToTemplate`** is called, which initializes the iteration to find the best matching template for our box. **`matchToTemplate`** takes one of the image templates as an argument, and is stored as `img_object`. At the same time, **`imageCallback`** captures a frame and stores it as `img`. Afterwards, `cv::cvtColor` is utilized to convert `img` into grayscale, in order to improve the matching accuracy. Similarly, `cv::resize` is called to change the aspect ratio of `img_object` to (500,400), approximately matching that of **`img`**. This is also done in order to improve the matching accuracy.

Next, the OpenCV SURF feature detector, `SURF::create(minHessian)`, is used to identify 400 (`=minHessian`) key points from both `img` and `img_object`, and then obtain descriptors which will be used for matching. With the descriptors ready, `DescriptorMatcher::FLANNBASED` is called to apply a FLANN matcher. This is a nearest neighbour keypoint matcher, which filters them based on the distance between them. In this case, the matches are filtered using the Lowe's Ratio Test [6], with a threshold of 0.75. This means that for a match to pass the test, the ratio between the distance of the closest match from the calculated point, and the distance of the second-closest match to that same point, must be below that threshold. Namely, in this case, we require our best match to be at least 1.33 times better than the second-best. If a match passes the test, it is stored in the `good_matches` vector.

Next, we use the **`findHomography`** function to create a matrix that will help us map the descriptors in our `img` with the ones in `img_object`, localizing it in our scene. Then **`perspectiveTransform`** is used in order to define the scene img corners. We use these corners with **`contourArea`**, obtaining the mapped `img` area. With this value, we apply an additional filter, if the area is greater than 5x5, and less than 1000x1000, then it sets `area_weight = 1`. Else, it sets `area_weight = 0`. In a later step, we will multiply this weight to the number of matches between `img` and `img_object`, effectively discarding images that are unlikely to be accurate, as they are too far from the box's dimensions. Next, we perform a final test, where we take each match stored in `good_matches`, and see if it is inside of the contour defined previously. If it is, then it is appended in the `best_matches` vector, if not, it is discarded. Finally, we count the number of elements in `best_matches`, multiply it by our previously calculated weight, and return it as the total number of matches of the box with that particular template.

The process outlined above is repeated for each box template. Within **`getTemplateID`**, two elements are initialized: `best_matches = 35` and `template_id = -1`. For each iteration, the output of **`matchToTemplate`** is compared to `best_matches`. If it is higher, it replaces it, and `template_id` is set to the index of the corresponding template. If no matches are higher than 35, the initial value, it is determined that the box is blank, and the `template_id` is unchanged. The best matching id is appended to the `IDHistory` list, which is used to check for duplicates. Finally, the **`getTemplateID`** function returns the id of the matched template, and writes it in our output file, along with the coordinates, the name of the template image file, and whether it is a duplicate or not, as mentioned in the contest requirements in Section 2.2.

5.2 Low-Level Control Architecture

While Section 5.1 focused on the high level path planning and image recognition details, this section describes the lower level details for how navigation is implemented based on the planned path.

5.2.1 Navigation

Having solved the TSP, our navigation strategy is to first use the `checkPlan` function, which uses the `"move_base/NavfnROS/make_plan"` service, to check whether it is possible to navigate to a given point indexed by the first dimension of `nav_coords`. If it is possible, then we call `Navigation::moveToGoal` with that point.

If it is not possible, then we adjust the navigation point slightly in up to 10 iterations, and go to the first new navigation point for which a successful path was found using the `checkPlan` function. If none of the 10 extra iterations yields a valid plan, then that box is skipped and the procedure is repeated for navigating to the next box in `nav_coords`.

The 10 extra iterations and the order of the iterations are shown in Figure 3. The path labelled with '0' is the initial navigation attempt from `nav_coords`; `offset=0.4` m away and directly facing the front face of the box. The next two iterations also occur at `offset` m away, but at an angle of ± 20° from the front face of the box respectively. Successive iterations then try navigating at ± 30°, ± 40°, ± 50°, and ± 60° from the front face of the box. After any successful path is found for any of these iterations, `Navigation::moveToGoal` is called, and remaining iterations are not checked. Our navigation strategy successively flips between checking the right and left side of the box at different angles rather than checking an entire side at a time in order to minimize time spent having to check through iterations. Without knowing which side of the box is obstructed for navigation and which side is clear a priori, it makes sense to flip between checking the sides in order to spend less time on average going through iterations.

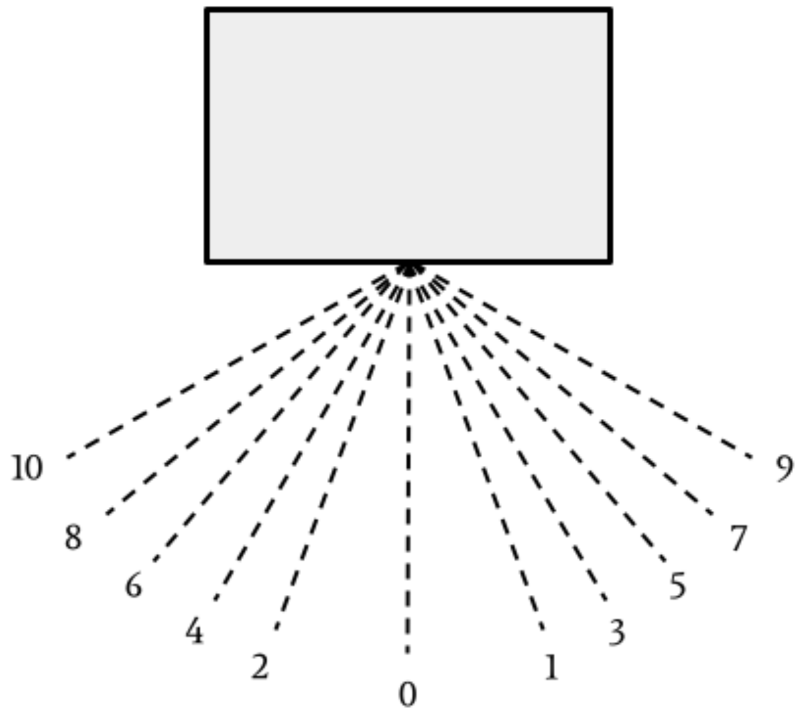

Figure 3: Top-down view of navigation iterations and corresponding positions

6.0 Future Recommendations

As described in Section 5.1 on High-Level Control Architecture, the program uses a brute force method to solve for the most efficient path of visiting all points of interest. In our program, as mentioned in Section 5.1.1.3, the algorithm checks all 10! possible paths. However, there is a more efficient implementation possible of our `bruteForceTSP` function, which would only need to check (9!)/2 paths rather than 10! paths.

Since we force the starting vertex of the cycle to be a particular value, we can fix that as a constraint in calculating the cycle weight and only look through the other 9! permutations of the vertices. Additionally, since the TurtleBot always returns to the starting point, and since the order of the cycle doesn't matter, we would only need to check half of the permutations. For example, a cycle of vertices (A, B, C, D, E, A), where our `bruteForceTSP` function would return corresponding path (A, B, C, D, E), is the same weight as cycle (A, E, D, C, B, A) with corresponding path (A, E, D, C, B). If we had done our implementation this way, then our `bruteForceTSP` function would only be looping through (9!)/2 = 181, 440 rather than 10! = 3, 628, 800 possible paths. Essentially, this strategy would allow us to find an optimal path within roughly the same time as if the environment had 11 boxes instead of

10, or if we decided to model the initial position of the TurtleBot as a graph vertex. To make the path planning algorithm adaptable to a greater number of boxes in the environment than this, we would likely need to implement the approximation algorithms discussed in Section 3.1.

In terms of image recognition, we are currently comparing the image the TurtleBot sees to the exact same image as the ground truth. This is relatively accurate for a small dataset where we can adjust the area of the bounding box, however, it would be unwieldy with a larger number of images. In future developments, a convolutional neural network (CNN) could be trained and then employed to determine the new image. CNNs have the added advantage that they are able to generalize, and even if the image it sees is slightly altered from its training set, it is still able to identify the category based on the extraction of key distinguishing features.

References

[1] T. Cormen and C. Leiserson, *Introduction to algorithms, 3rd edition*, 3rd ed.

[2] H. Ong and J. Moore, "Worst-case analysis of two travelling salesman heuristics", *Operations Research Letters*, vol. 2, no. 6, pp. 273-277, 1984. Available: 10.1016/0167-6377(84)90078-6.

[3] M. Soo, "Traveling Salesman Problem", *Github*, 2021. [Online]. Available: https://github.com/matteosoo/Traveling-Salesman-Problem. [Accessed: 25- Mar- 2021].

[4] G. Nejat and M. Effati, "Tutorial 4 - OpenCV Features", University of Toronto, 2021.

[5] "OpenCV: Features2D + Homography to find a known object", *Docs.opencv.org*, 2021. [Online]. Available: https://docs.opencv.org/3.4.4/d7/dff/tutorial_feature_homography.html. [Accessed: 25- Mar- 2021].

[6] "OpenCV: Feature Matching with FLANN", *Docs.opencv.org*, 2021. [Online]. Available: https://docs.opencv.org/3.4/d5/d6f/tutorial_feature_flann_matcher.html. [Accessed: 25- Mar- 2021].

[7] "Travelling Salesman Problem (Basics + Brute force approach)", *OpenGenus IQ: Learn Computer Science*, 2021. [Online]. Available: https://iq.opengenus.org/travelling-salesman-problem-brute-force/. [Accessed: 25- Mar- 2021].

Appendices

Appendix A - Complete C++ Code - `contest2.cpp`

Full code also available at: https://github.com/farhanwadia/mie443_contest2

```
#include <boxes.h>
#include <navigation.h>
#include <robot_pose.h>
#include <imagePipeline.h>
#include <chrono>
#include <nav_msgs/GetPlan.h>
#include <iostream>
#include <fstream>
#include <experimental/filesystem>
#include <ctime>
#include <locale>
#include <vector>
#include <algorithm>
#include <string>

#define RAD2DEG(rad) ((rad)*180./M_PI)
#define DEG2RAD(deg) ((deg)*M_PI/180.)

float dist(float x1, float y1, float x2, float y2){
    //Calculates the Euclidean distance between two points (x1, y1), (x2, y2)
    return sqrt(pow(x2-x1, 2) + pow(y2-y1, 2));
}

float add2Pi(float angle){
    //Converts angles from -pi to 0 to be between pi and 2pi
    if (angle < 0){
        angle = angle + 2*M_PI;
    }
    return angle;
}

float minus2Pi(float angle){
    //Converts angles from pi to 2pi to be between -pi and 0
    if(angle > M_PI){
        angle = angle - 2*M_PI;
    }
    return angle;
}
void fillNavCoords(float nav_coords[10][3], Boxes* pBoxes, float offset){
    //Fills in the nav_coords array using boxes.coords
    //New x,y,z are calculated such that the turtlebot faces directly in front of the
box at offset distance away
    Boxes boxes;
    boxes = *pBoxes;
    for(int i=0; i < boxes.coords.size(); i++){
        nav_coords[i][0] = boxes.coords[i][0] + offset*cosf(boxes.coords[i][2]); //set
x value
        nav_coords[i][1] = boxes.coords[i][1] + offset*sinf(boxes.coords[i][2]); //set
y value
        nav_coords[i][2] = minus2Pi(boxes.coords[i][2] + M_PI); //set angle
    }
}

void fillAdjacencyMatrix(float adjMat[10][10], float nav_coords[10][3]){
    //Fills adjacency matrix of graph edge weights (i.e. distance between nodes i, j)
    for(int i=0; i<10; i++){
        for(int j=0; j<10; j++){
            adjMat[i][j] = dist(nav_coords[i][0], nav_coords[i][1], nav_coords[j][0],
nav_coords[j][1]);
            ROS_INFO("Distance (%d, %d): %.3f", i, j, adjMat[i][j]);
        }
    }
}

int findClosestBoxAtStart(float nav_coords[10][3]){
    // Returns the index of the box closest to the start location (0, 0)
    float minD = std::numeric_limits<float>::infinity();
    float d;
    int argmin;
    for(int i=0; i<10; i++){
        d = dist(0, 0, nav_coords[i][0], nav_coords[i][1]);
        ROS_INFO("Distance to node %d: %.3f", i, d);
        if (d < minD){
            minD = d;
            argmin = i;
        }
    }
    return argmin;
}
```

```
float bruteForceTSP(float nav_coords[10][3], float adjMat[10][10], int source,
std::vector<int> &TSPTour){
    //Returns the distance of the optimal TSP tour and modifies vector TSPTour to
provide the nodes in the order of the tour path
    //Adapted from https://iq.opengenus.org/travelling-salesman-problem-brute-force/
    std::vector<int> nodes;
    int num_nodes = 10;

    //Append the other nodes to the vector
    for(int i=0; i<num_nodes; i++){
        if(i != source){
            nodes.push_back(i);
        }
    }
    int n = nodes.size();
    float shortestPathWgt = std::numeric_limits<float>::infinity();

    //Generate permutations and track the minimum weight cycle
    while(next_permutation(nodes.begin(), nodes.end())){
        float currentPathWgt = 0;
        std::vector<int> currentTour;

        int j = source;
        currentTour.push_back(source);

        //Calculate distance and visiting order for current tour path
        for (int i = 0; i < n; i++)
        {
            currentPathWgt += adjMat[j][nodes[i]];
            j = nodes[i];
            currentTour.push_back(j);
        }
        currentPathWgt += adjMat[j][source]; //add the distance from last node back to
source

        //Update shortest path and the node order if the current tour is smaller than
the previous minimum
        if (currentPathWgt < shortestPathWgt){
            shortestPathWgt = currentPathWgt;
            TSPTour = currentTour;
        }
    }
    ROS_INFO("TSP Distance: %.5f \n Printing Node Order:", shortestPathWgt);
    for(int i=0; i<TSPTour.size(); i++){
        ROS_INFO("%d", TSPTour[i]);
    }
    return shortestPathWgt;
}

bool checkPlan(ros::NodeHandle& nh, float xStart, float yStart, float phiStart, float
xGoal, float yGoal, float phiGoal){
       //Returns true if there is a valid path from (xStart, yStart, phiStart) to
(xGoal, yGoal, phiGoal)
    //Adapted from
https://answers.ros.org/question/264369/move_base-make_plan-service-is-returning-an-emp
ty-path/

    bool callExecuted, validPlan;

    //Set start position
    geometry_msgs::PoseStamped start;
    geometry_msgs::Quaternion phi1 = tf::createQuaternionMsgFromYaw(phiStart);
    start.header.seq = 0;
    start.header.stamp = ros::Time::now();
    start.header.frame_id = "map";
    start.pose.position.x = xStart;
    start.pose.position.y = yStart;
    start.pose.position.z = 0.0;
    start.pose.orientation.x = 0.0;
    start.pose.orientation.y = 0.0;
    start.pose.orientation.z = phi1.z;
    start.pose.orientation.w = phi1.w;

    //Set goal position
    geometry_msgs::PoseStamped goal;
    geometry_msgs::Quaternion phi2 = tf::createQuaternionMsgFromYaw(phiGoal);
    goal.header.seq = 0;
    goal.header.stamp = ros::Time::now();
    goal.header.frame_id = "map";
    goal.pose.position.x = xGoal;
    goal.pose.position.y = yGoal;
    goal.pose.position.z = 0.0;
    goal.pose.orientation.x = 0.0;
    goal.pose.orientation.y = 0.0;
    goal.pose.orientation.z = phi2.z;
    goal.pose.orientation.w = phi2.w;

    //Set up the service and call it
    ros::ServiceClient check_path =
nh.serviceClient<nav_msgs::GetPlan>("move_base/NavfnROS/make_plan");
    nav_msgs::GetPlan srv;
    srv.request.start = start;
    srv.request.goal = goal;
    srv.request.tolerance = 0.0;
    callExecuted = check_path.call(srv);

    if(callExecuted){
        ROS_INFO("Call to check plan sent");
    }
    else{
        ROS_INFO("Call to check plan NOT sent");
    }

    if(srv.response.plan.poses.size() > 0){
        validPlan = true;
        ROS_INFO("Successful plan of size %ld", srv.response.plan.poses.size());
    }
    else{
        validPlan = false;
        ROS_INFO("Unsuccessful plan");
    }
    return validPlan;
}

bool isDuplicate(std::vector<int> &IDHistory, int template_id){
    //Check if template_id already exists in the IDHistory
    bool duplicate = false;
    if(std::find(IDHistory.begin(), IDHistory.end(), template_id) !=IDHistory.end()){
        duplicate = true;
    }
    return duplicate;
}

std::string tagIndexToString(int idx){
    std::string label = "tag_";
    if(idx == -1){
        label = label + "blank" + ".jpg";
    }
    else{
        label = label + std::to_string(idx+1) + ".jpg";
    }
    return label;
}
```

```
int main(int argc, char** argv) {
    // Setup ROS.
    ros::init(argc, argv, "contest2");
    ros::NodeHandle n;
    // Robot pose object + subscriber.
    RobotPose robotPose(0,0,0);
    ros::Subscriber amclSub = n.subscribe("/amcl_pose", 1, &RobotPose::poseCallback,
&robotPose);
    // Initialize box coordinates and templates
    Boxes boxes;
    if(!boxes.load_coords() || !boxes.load_templates()) {
        std::cout << "ERROR: could not load coords or templates" << std::endl;
        return -1;
    }
    // Initialize image object and subscriber.
    ImagePipeline imagePipeline(n);

    float adjMat[10][10];
    float nav_coords[10][3];
    int startBox, currentNode = 0, template_id;
    float xx, yy, zz, dz, offset = 0.4, TSPDist;
    bool nav_success, valid_plan, duplicate_check = false;
    std::vector<int> TSPTour, IDHistory;
    std::string duplicateText, filename, timeStr;

    //Contest count down timer
    std::chrono::time_point<std::chrono::system_clock> start;
    start = std::chrono::system_clock::now();
    uint64_t secondsElapsed = 0;

    //Get timestamp for output file name
    time_t t = time(0);   // get time now
    struct tm * now = localtime(&t);

    char timestamp [150];
    strftime (timestamp, 150,"%Y-%m-%d %H-%M-%S", now);
    timeStr = std::string(timestamp);
    filename = std::experimental::filesystem::current_path() +
"/src/mie443_contest2/Group 18 Output - " + timeStr + ".csv";

    //Write output file if it doesn't exist yet with headers
    std::ofstream output(filename);
    output << "\"Order Visited\"" << ", " << "\"Box ID\"" << ", " << "\"Tag File\"" <<
", "
```

```
    << "\"Discovered as Duplicate\"" << ", " << "\"Location (x, y, angle)\"" << 
std::endl;

    //Fill the nav_coords array
    fillNavCoords(nav_coords, &boxes, offset);
    for(int i =0; i < 10; i++){
        ROS_INFO("Box %d: (%.3f, %.3f, %.3f)", i, boxes.coords[i][0], 
boxes.coords[i][1], boxes.coords[i][2]);
        ROS_INFO("Nav %d: (%.3f, %.3f, %.3f) \n", i, nav_coords[i][0], 
nav_coords[i][1], nav_coords[i][2]);
    }

    //Fill adjacency matrix of graph edge weights (i.e. distance between nodes i, j)
    fillAdjacencyMatrix(adjMat, nav_coords);

    //Find closest box to current location. Start the tour from there
    startBox = findClosestBoxAtStart(nav_coords);
    ROS_INFO("Start Box: %d", startBox);

    //Brute Force TSP. TSPTour is the path corresponding to the 10 node TSP cycle
    TSPDist = bruteForceTSP(nav_coords, adjMat, startBox, TSPTour);

    //Execute strategy.
    while(ros::ok() && secondsElapsed <= 480) {
        ros::spinOnce();
        /***YOUR CODE HERE***/
        // Use: boxes.coords
        // Use: robotPose.x, robotPose.y, robotPose.phi
        //imagePipeline.getTemplateID(boxes);

        //Travel to the nodes and then back to the start
        if (currentNode <= 10){
            if (currentNode < 10){
                xx = nav_coords[TSPTour[currentNode]][0];
                yy = nav_coords[TSPTour[currentNode]][1];
                zz = nav_coords[TSPTour[currentNode]][2];
            }
            else {
                //Return to start after exploring all nodes
                xx = 0;
                yy = 0;
                zz = 0;
            }

            ROS_INFO("Testing TSP node %d (original %d). (%.3f, %.3f, %.3f)",
currentNode, TSPTour[currentNode], xx, yy, zz);
            valid_plan = checkPlan(n, robotPose.x, robotPose.y, robotPose.phi, xx, yy,
zz);

            //Try varying the angle to be +/- 20, 30, 40, 50, 60 deg from centre if
navigation was unsuccesful
            dz = DEG2RAD(20);
            while(!valid_plan && fabs(dz) <=DEG2RAD(61) && currentNode < 10){
                // Recalculate xx, yy, zz, to incorporate angle offset dz
                xx = boxes.coords[TSPTour[currentNode]][0] +
offset*cosf(boxes.coords[TSPTour[currentNode]][2] + dz);
                yy = boxes.coords[TSPTour[currentNode]][1] +
offset*sinf(boxes.coords[TSPTour[currentNode]][2] + dz);
                zz = minus2Pi(boxes.coords[TSPTour[currentNode]][2] + dz + M_PI);
                //Try new plan
                ROS_INFO("Testing TSP node %d (original %d) with offset %.1f. (%.3f,
%.3f, %.3f)", currentNode, TSPTour[currentNode], RAD2DEG(dz), xx, yy, zz);
                valid_plan = checkPlan(n, robotPose.x, robotPose.y, robotPose.phi, xx,
yy, zz);
                if(valid_plan){
                    break;
                }
                //Change dz if navigation still unsuccesful
                if(dz > 0){
                    dz = -dz;
                }
                else{
                    dz = -dz;
                    dz = dz + DEG2RAD(10);
                }
            }

            //Navigate if the path plan is valid
            if (valid_plan){
                nav_success = Navigation::moveToGoal(xx, yy, zz, 30.0);
                ROS_INFO("Finshed moving. Nav Status: %d", nav_success);
                if(nav_success && currentNode < 10){
                    //Add wait time to ensure correct image capture
                    ros::Duration(0.10).sleep();
                    ros::spinOnce();
                    ros::Duration(0.60).sleep();
                    ros::spinOnce();

                    int template_id = imagePipeline.getTemplateID(boxes);
                    ROS_INFO_STREAM("Match: " << tagIndexToString(template_id));
                    duplicate_check = isDuplicate(IDHistory, template_id);

                    //Append template_id to a vector called IDHistory if not already 
there
                    if(duplicate_check){
                        duplicateText = "True";
                    }
                    else{
                        IDHistory.push_back(template_id);
                        ROS_INFO("Appended %i to IDHistory", template_id);
                        duplicateText = "False";
                    }
                    //Discovery Order; Box #; Tag ID; Is Duplicate; Location 
Coordinates;
                    ROS_INFO("Appending to output file");
                    output << "\"" << currentNode+1 << "\", \"" << TSPTour[currentNode] 
<< "\", \"" << tagIndexToString(template_id) << "\", \"" << duplicateText << "\", \"" 
<< "("
                    << boxes.coords[TSPTour[currentNode]][0] << ", " << 
boxes.coords[TSPTour[currentNode]][1] << ", " << boxes.coords[TSPTour[currentNode]][2] 
<<  ")" << "\""
                    << std::endl;
                }
                else{
                    if(currentNode < 10){
                        ROS_INFO("PLAN VALID BUT NAVIGATION FAILED");
                        output << "\"" << currentNode+1 << "\", \"" << 
TSPTour[currentNode] << "\", \"" << "" << "\", \"" << "" << "\", \"" << "\"" << 
std::endl;
                    }
                }
            }
            if (fabs(dz) > DEG2RAD(61) || !valid_plan){
                ROS_INFO("COULD NOT FIND ANY PATH TO NODE %d", currentNode);
            }

            ROS_INFO("Elapsed time %ld", (long)secondsElapsed);
            currentNode ++;
        }
        else{
            //Explored all 10 nodes and returned to start
            break;
        }
        secondsElapsed =
std::chrono::duration_cast<std::chrono::seconds>(std::chrono::system_clock::now()-start
).count();
        ros::Duration(0.01).sleep();
    }
    return 0;
}
```

Appendix B - Complete C++ Code - `imagePipeline.cpp`

```
#include <imagePipeline.h>
#include <string>

#define IMAGE_TYPE sensor_msgs::image_encodings::BGR8
#define IMAGE_TOPIC "camera/rgb/image_raw" // kinect:"camera/rgb/image_raw"
webcam:"camera/image"

ImagePipeline::ImagePipeline(ros::NodeHandle& n) {
    image_transport::ImageTransport it(n);
    sub = it.subscribe(IMAGE_TOPIC, 1, &ImagePipeline::imageCallback, this);
    isValid = false;
}

void ImagePipeline::imageCallback(const sensor_msgs::ImageConstPtr& msg) {
    try {
        if(isValid) {
            img.release();
        }
        img = (cv_bridge::toCvShare(msg, IMAGE_TYPE)->image).clone();
        isValid = true;
    } catch (cv_bridge::Exception& e) {
        std::cout << "ERROR: Could not convert from " << msg->encoding.c_str()
                  << " to " << IMAGE_TYPE.c_str() << "!" << std::endl;
        isValid = false;
    }
}

double ImagePipeline::matchToTemplate(Mat img_object){
    //convert image to grayscale
    cv::Mat gray_img;
    cv::cvtColor(img, gray_img, cv::COLOR_BGR2GRAY); // Convert scene image to
greyscale to improve accuracy
    cv::resize(img_object, img_object, cv::Size(500,400)); // Resize tag image to match
scene aspect ratio

    //--Step 1 & 2: Detect the keypoints and calculate descriptors using SURF Detector
    int minHessian = 400;
    Ptr<SURF> detector = SURF::create(minHessian);
    std::vector<KeyPoint>keypoints_object, keypoints_scene;
    Mat descriptors_object, descriptors_scene;
    detector->detectAndCompute(img_object, Mat(), keypoints_object,
descriptors_object);
    detector->detectAndCompute(gray_img, Mat(), keypoints_scene, descriptors_scene);
    //Lowe's ratio filer
    //-- Step 3: Matching descriptor vectors using FLANN matcher
    Ptr<DescriptorMatcher> matcher = 
DescriptorMatcher::create(DescriptorMatcher::FLANNBASED);
    std::vector< std::vector<DMatch> > knn_matches;
    matcher->knnMatch( descriptors_object, descriptors_scene, knn_matches, 2 );

    //-- Filter matches using the Lowe's ratio test
    const float ratio_thresh = 0.75f;
    std::vector<DMatch> good_matches;
    for (size_t i = 0; i < knn_matches.size(); i++) {
        if (knn_matches[i][0].distance < ratio_thresh * knn_matches[i][1].distance){
            good_matches.push_back(knn_matches[i][0]);
        }
    }

    //-- Localize the object
    std::vector<Point2f> obj;
    std::vector<Point2f> scene;

    for( int i = 0; i < good_matches.size(); i++ )
    {
    //-- Get the keypoints from the good matches
    obj.push_back( keypoints_object[ good_matches[i].queryIdx ].pt );
    scene.push_back( keypoints_scene[ good_matches[i].trainIdx ].pt );
    }

    Mat H = findHomography( obj, scene, RANSAC );

    //-- Get the corners from the img_object ( the object to be "detected" )
    std::vector<Point2f> obj_corners(4);
    obj_corners[0] = cvPoint(0,0);
    obj_corners[1] = cvPoint( img_object.cols, 0 );
    obj_corners[2] = cvPoint( img_object.cols, img_object.rows );
    obj_corners[3] = cvPoint( 0, img_object.rows );
    std::vector<Point2f> scene_corners(4);

    // Define scene_corners using Homography
    try {
        perspectiveTransform(obj_corners, scene_corners, H);
    } catch (Exception& e) {
        ;
    }
    // Define contour using the scene_corners
    std::vector<Point2f> contour;
    for (int i = 0; i < 4; i++){
           contour.push_back(scene_corners[i] + Point2f( img_object.cols, 0));
    }
    double area = contourArea(contour);
    double area_weight = 1.0;
    if (area >  1000*1000 || area < 5 * 5)
    {
        area_weight = 0.0;
    }
    else
    {
        area_weight = 1.0;
    }
    std::cout << "area: " << area << ", weight: " << area_weight << std::endl;

    double indicator;
    std::vector< DMatch > best_matches;

    // Check if the good match is inside the contour. If so, write in best_matches and
multiply by area weight
    Point2f matched_point;
    for( int i = 0; i < good_matches.size(); i++ )
    {
        matched_point = keypoints_scene[ good_matches[i].trainIdx ].pt + Point2f(
img_object.cols, 0);
        indicator = pointPolygonTest(contour, matched_point, false);
        if(indicator >= 0) best_matches.push_back( good_matches[i]);
    }

    cv::waitKey(10);

    return (double)best_matches.size()*area_weight;
}

int ImagePipeline::getTemplateID(Boxes& boxes) {
    int template_id = -1;
    double best_matches, matches;
    if(!isValid) {
        std::cout << "ERROR: INVALID IMAGE!" << std::endl;
    }
    else if(img.empty() || img.rows <= 0 || img.cols <= 0) {
        std::cout << "ERROR: VALID IMAGE, BUT STILL A PROBLEM EXISTS!" << std::endl;
        std::cout << "img.empty():" << img.empty() << std::endl;
        std::cout << "img.rows:" << img.rows << std::endl;
        std::cout << "img.cols:" << img.cols << std::endl;
    }
    else {
        // Initialize best match counter. If less than 35, it is likely to be blank
        best_matches = 35;
        template_id = -1;

        // Test against each box template
        for (int i = 0; i < boxes.templates.size(); ++i)
        {
            // Get the number of matches between the scene and box template
            matches = matchToTemplate(boxes.templates[i]);
            std::cout  << matches << " matching features for template " << i <<
std::endl;

            // Save the maximum matches and corresponding box index
            if (matches > best_matches){
                best_matches = matches;
                template_id = i;
            }
        }
    }
    // Get grayscale image to view in RVIZ
    cv::Mat gray_img;
    cv::cvtColor(img, gray_img, cv::COLOR_BGR2GRAY);

    // Display the scene image
    cv::imshow("view", gray_img);
    cv::waitKey(1000);
    std::cout  << "Best template is  " << template_id << " with " << best_matches << "
matches" << std::endl;

    return template_id;
}
```

Appendix C - Complete C++ Code - `navigation.cpp`

```
#include <navigation.h>
#include <actionlib/client/simple_action_client.h>
#include <move_base_msgs/MoveBaseAction.h>
#include <tf/transform_datatypes.h>

bool Navigation::moveToGoal(float xGoal, float yGoal, float phiGoal, float timeout){
       // Set up and wait for actionClient.
    actionlib::SimpleActionClient<move_base_msgs::MoveBaseAction> ac("move_base",
true);
    while(!ac.waitForServer(ros::Duration(5.0))){
        ROS_INFO("Waiting for the move_base action server to come up");
    }
       // Set goal.
    geometry_msgs::Quaternion phi = tf::createQuaternionMsgFromYaw(phiGoal);
    move_base_msgs::MoveBaseGoal goal;
    goal.target_pose.header.frame_id = "map";
    goal.target_pose.header.stamp = ros::Time::now();
    goal.target_pose.pose.position.x =  xGoal;
    goal.target_pose.pose.position.y =  yGoal;
    goal.target_pose.pose.position.z =  0.0;
    goal.target_pose.pose.orientation.x = 0.0;
    goal.target_pose.pose.orientation.y = 0.0;
    goal.target_pose.pose.orientation.z = phi.z;
    goal.target_pose.pose.orientation.w = phi.w;
    ROS_INFO("Sending goal location ...");
       // Send goal and wait for response.
    ac.sendGoal(goal);
    ac.waitForResult(ros::Duration(timeout));
    if(ac.getState() == actionlib::SimpleClientGoalState::SUCCEEDED){
        ROS_INFO("You have reached the destination");
        return true;
    } else {
        ROS_INFO("The robot failed to reach the destination");
        return false;
    }
}
```

Appendix D - Attribution Table

Category		**Farhan**	**Henry**	**Yilin**
Code	Strategy	ET	RD	
	Handler Functions	RD	RD	RD
	High-level Strategy Functions	RD	ET	
	Integration	ET	ET	
	Clean-up		RD	
	Testing	FP	FP	FP
Report	FOCs			RD
	Strategy	RS, RD	RS, RD	
	Sensors			RD
	Controller	RS, RD	RS, RD	
	Complete Report	ET	ET	ET, CM

RS	Research
RD	Wrote first draft
MR	[CODE] Major revision to strategy, functions used [REPORT] Major revision to organization
ET	[CODE] Edited for errors, de-bugging [REPORT] Edited for grammar and spelling
FP	[CODE] Final check for applicability to practice worlds [REPORT] Final check for flow and consistency
CM	Responsible for compiling the elements into the complete document