Creating a Robotic Waiter using a Pioneer P3DX Robot

Beverly:
James Adey, James Atkin, Fraser Brooks, Thomas Taylor, Barney Whiteside

***Abstract*— In this paper, a set of ROS [1] nodes are programmed to talk to a P3DX robot which models the behaviour of a waiter, moving to tables with an order when instructed with a mobile app, and then scanning a QR code which confirms the received order. These actions required pathfinding and localisation implementations which were programmed in C++ and Python without any help from external libraries besides the QR code scanning for image parsing. The following experiments were also performed:**

- **The effect on pathfinding with node count: the results of which pointed towards a node count of around 1500 being the most optimal for path cost.**
- **Travel time vs map node count: the results of which showed that due to local pathfinding and the robot's speed, the overall time to reach a goal is not affected by node count.**

I. INTRODUCTION

Robotics is a field which has already demonstrated the capacity for intelligent machines that can help humans in many real, tangible ways. One task which is not widely performed by robots, however, is that of waiting in a restaurant or bar. This is a complex task that requires a robot take orders, navigate to customers, and avoid obstacles in busy environments. This task is unique compared to other more explored robot tasks such as warehouse worker since it cannot be assumed that the path will be unobstructed and there must be reliance on user input for correct allocation of jobs and for manual confirmation of job completion. These parameters mean that the most important features of the project are that of dynamic path planning and obstacle avoidance within a mapped area and that of customer interaction via an app for ordering and QR code scanning or job completion. If either of these parts fail then the robot is not suitable for purpose. You will see through our choices in algorithms and system design that the priorities when approaching the task were to make the system robust and reliable.

II. RELATED WORK

The navigation stack [2] is a completed codebase that contains methods that cover the motion and navigation of any robots running ROS. This framework is specific to rectangular and circular robots as these were the design that the nav-stack was originally developed on. The purpose of the nav-stack is to be as generic as possible and facilitate navigation and motion in any given project.

While the implementation in this project uses a graph generated for the map, the nav-stack is able to both map and navigate in a given area. It does not make guarantees about optimal path distances and only has two levels of navigation with global points of interest and local obstacle avoidance. This project's implementation takes this a step further with dynamic path re-planning (used alongside local obstacle avoidance). This is described in algorithmic detail in Section III-E.2.

The code repository [**?**] for the ROS nav-stack details the functions covered.

A paper written by Asif, M & Sabeel, M & Khan, Zeashan details an attempt to implement a robot waiter in a lab setting and mock restaurant [3]. The attempt is successful in their result metric which is customer satisfaction. They focus much more on the Human Computer Interaction end of restaurant automation in which they describe the layout and functionality of their LCD screen system that customers use to interact with the robot and they do not consider the problem of path planning and obstacle avoidance in any great detail. Instead they use floor line tracking to allow the robot to navigate the space and make the assumption that there will not be any dynamic obstacles to worry about in their simplified example.

III. ALGORITHMS AND FRAMEWORKS

A. Overview of the system

The Robot Waiter consists of 4 main subsystems, each responsible for a separate portion of the functionality. This decision was made partially for ease of software engineering allowing each subsystem to be developed in parallel, but also to permit subsystems to be swapped out and exchanged with alternative implementations with minimal effort.

- **Orders Brain**: Responsible for receiving, fulfilling and managing orders, as well as broadcasting the current goal pose for the robot.
- **Localisation**: Responsible for calculating and broadcasting the robot's position relative to the map.
- **Path Finding**: Responsible for calculating and consistently broadcasting global path to the current goal pose.
- **Movement**: Responsible for moving the robot around the world, following paths and performing localisation motion when necessary.

B. Localisation

The particle filter created in Assignment 2 was used for the robot's localisation. Improvements were made to make it more reliable and the pose estimate was modified to give a more accurate estimate. This was important as the pose estimate was required by other components and if it was wrong this could have dire consequences for the success of the system.

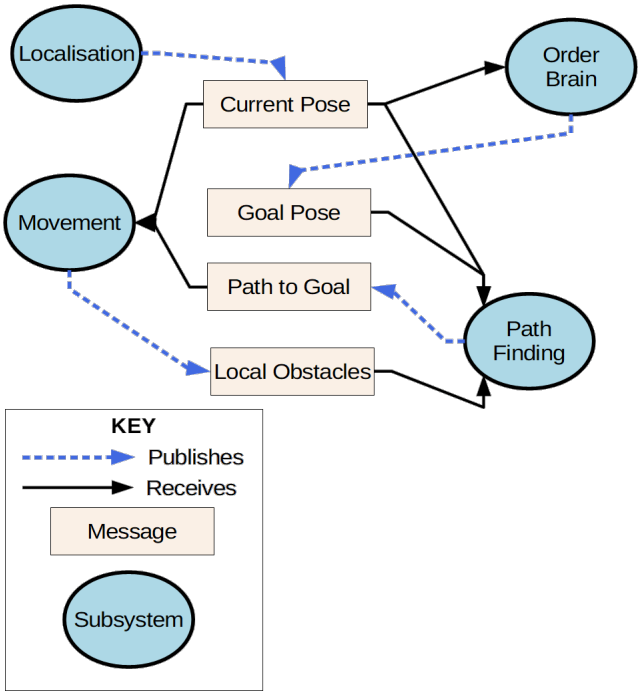

Fig. 1. A High-level overview of the robot waiter sub-systems and how they interact with one another

The changes to the algorithm involved adding adaptive re-sampling, such that fewer particles are uniformly re-sampled as the pose certainty increases. The pose certainty is defined as the proportion of the total weight that exists in the largest cluster of particles.

Once a certainty limit (in our case 90%) has been reached, no particles are uniformly sampled. So whilst the certainty exceeds this limit, future generations of particles are entirely sampled (in proportion to the weights) from the previous generation.

In addition, the pose certainty was included with the output message, permitting other systems to detect when the robot is localised. This could be a covariance matrix, but in this project the certainty calculated was sent as a floating point value.

C. Pathfinding

For the pathfinding, a ROS node was created to run on the machine, finding a path from the robot's pose estimate to the goal destination. This node runs continually, publishing on the /`path` topic and its output can be accepted or ignored by other components.

1) Building a graph: To construct the graph, free open-source 3D modelling software Blender was used to create a plane of the map, divided into triangular regions that the robot could move through. These were then exported in .obj format, and read into the system which creates a traversable graph with nodes at the centre of each triangle, while preserving the adjacencies. This map was modified several times throughout the project, moving nodes away from walls if the robot was too close, and adding more triangles to the map to create more cycles.

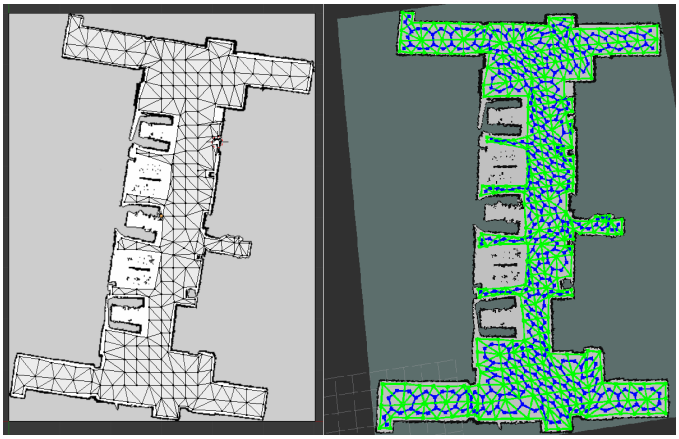

Fig. 2. Left: Map as shown in Blender; Right: RViz visualisation of triangle map (green) and generated graph (blue)

2) Finding a path over the graph: An A* search algorithm was implemented for pathfinding over the graph as it is efficient and reasonably lightweight. The heuristic was the Euclidean distance from the current node to the destination. As this is guaranteed to be less than or equal to the true distance, this algorithm is admissible and guaranteed to return the optimal path.

The pathfinder subscribes to the topics /`amcl_pose` and /`move_base_simple/goal` to get the pose estimate and goal destination, respectively.

One level of obstacle detection was also implemented in the pathfinding, however this is explained later in III-E.2.

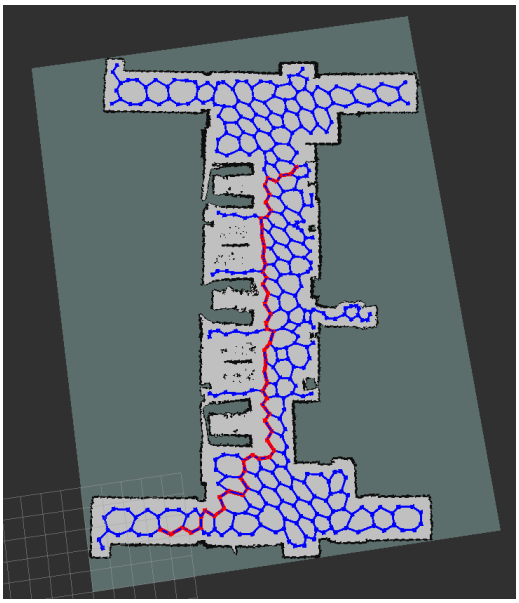

Fig. 3. RViz visualisation of path (red) found over generated graph (blue)

D. Movement

This section details the movement node in the system. When moving around the world,

Algorithm 1: Motion controller state machine

Input: pose certainty factor C, upper and lower thresholds T_U, T_L

```
if C > T_U then
    localised ← true
else if C < T_L then
    localised ← false
Locate obstacles
if localised then
    Clear all obtacles from detected empty space
    Mark all detected obstacles
    Inflate all obstacles
    if followMode = GLOBAL then
        Follow the global path
    else if followMode = LOCAL then
        Follow the local path
else
    Perform localisation motion
```

1) Motion controller "state" machine: *Algorithms for following the global and local paths are included below. Our implementation uses a simple wall-hugger for the localisation motion, but any suitable procedure for automating localisation is sufficient, algorithms for implementing a wall-huggers are well known, and thus is not included in this report.*

2) Defining Obstacles: The robot has a global cost-map structure, of the same resolution and size as the occupancy grid specified by the world map. This is utilised by the local pathfinding for obstacle avoidance.

The current laser scan result is filtered for hits and misses. A set of detected obstacles are created from the hits. Represented as a 6-tuple of $< d, \theta, L_x, L_y, W_x, W_y >$ where d is the local distance from robot, θ is the relative angle from robot's current facing, L_x and L_y are the computed coordinates of the obstacle in the robots frame, W_x and W_y are the computed coordinates of the obstacle in the world frame.

Various operations are required to maintain the accuracy of the cost-map, marking and clearing obstacles, as well as marking inflated cells and computing obstacle closeness.

Firstly, marking obstacles on the map involves converting the all of the detected obstacles world positions into map-cells, then flagging these cells as *true obstacles*.

Cells are cleared by raytracing each laser miss against the cost-map for a certain distance, clearing the obstructed flag.

Marking inflated cells and computing obstacle closeness is done in 2 passes. Firstly all closeness and inflation is cleared in the first pass. Then another pass is performed, where for each cell, if it is a true obstacle, it is flagged as obstructed. If the cell is obstructed then an inflation filter is applied.

An inflation filter can be applied to a certain cell and affects all surrounding cells, flagging them as obstructed if close enough, and assigning a higher closeness value if necessary.

The binary inflation filter and closeness filter are both pre-computed at start-up and re-used for efficiency. Represented as an array of pairs of $< inflate, closeness >$ where inflate is a boolean value and closeness a floating point value between 0 and 127.

3) Global Path Following: Whilst following the global path the robot attempts to optimise it's motion by looking ahead for clear accessible nodes that it can get to.

The below algorithm uses the term "trace robot hull", this means to trace a circle representing the robot's hull along a line and see if any local obstacles intersect this. This procedure is explained further into the paper.

Algorithm 2: Global Path Following

Input: robot position R, set of dynamic obstacles O, path of n nodes P, maximum global look-ahead distance L_g, inflation map M

```
lastClearNode ← −1
i ← 0
for i to n do
    d ← distance from R to P_i
    if d > L_g then
        break
    Trace robot hull from R to P_i against O
    if no obstacle found then
        Trace line from R to P_i against M
        if line doesn't intersect inflation then
            lastClearNode = i
end
localPathEnd ← (i − 1)
blocked ← (lastClearNode = 0 and n > 1)
else if lastClearNode = -1 or blocked then
    Switch to local path following
      followMode ← LOCAL
    localPathEndPose ← P_localPathEnd
    localGoalPose ← P_0
localGoalPose ← P_lastClearNode
Move in a straight line towards localGoalPose
```

E. Efficient Robot Hull Tracing

Algorithm 3 presents an optimised procedure used when tracing the robot's hull against an obstacle using a distance to line segment calculation. To perform the hull trace, this procedure is repeated for all the obstacles and performs in linear time dependent on the number of obstacles.

1) Local Path Following:

2) Global Obstacle Avoidance: Obstacle avoidance was performed on multiple levels, at the global level this involves re-planning around obstacles to hopefully find an alternative route to the goal.

The robot would sometimes get stuck in front of an obstacle which blocked its path, with no way to proceed to the next node in its path. To fix this the pathfinder was modified, subscribing to "`local_obstacles`", and the ability was added to mark nodes as 'obstructed'. For

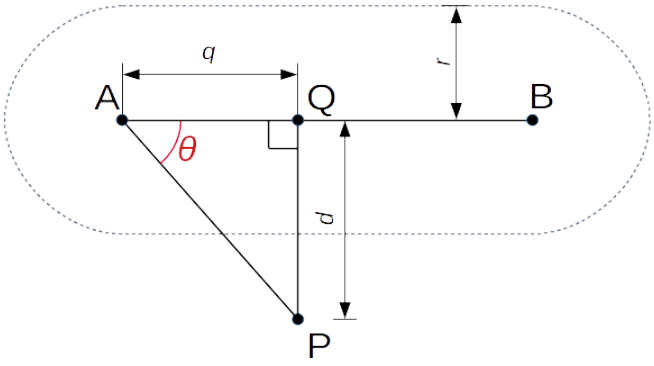

Fig. 4. Graphical representation of calculation of the distance between a point and a line segment

Algorithm 3: Optimised procedure for calculating distance between a point and line segment

Definitions: A,Q,P,B are points in space. Q is defined as the point on line segment AB that is closest to point P. Line AB is perpendicular to line QP.

Observing figure 4, the following 3 equations can be derived.

(1) $\cos\theta = \frac{\vec{AP}\cdot\vec{AB}}{\|\vec{AP}\|\|\vec{AB}\|}$

(2) $\|\vec{AQ}\| = \|\vec{AP}\|\cos\theta$

/* $adj = hyp \times \cos\theta$ */

/* Q exists at a proportion q along $\vec{AB}$ */

(3) $q = \frac{\|\vec{AQ}\|}{\|\vec{AB}\|}$

Combining equations (1) and (2), gives:

$\|\vec{AQ}\| = \|\vec{AP}\| \times \frac{\vec{AP}\cdot\vec{AB}}{\|\vec{AP}\|\|\vec{AB}\|}$

Which can be simplified to:

(4) $\|\vec{AQ}\| = \frac{\vec{AP}\cdot\vec{AB}}{\|\vec{AB}\|}$

Combining equations (3) with (4) gives: $q = \frac{\frac{\vec{AP}\cdot\vec{AB}}{\|\vec{AB}\|}}{\|\vec{AB}\|}$

Which can be simplified to: $q = \frac{\vec{AP}\cdot\vec{AB}}{\|\vec{AB}\|^2}$

If $q < 0$ or $q > 1$ then the point Q does not lie within the segment $\vec{AB}$. Therefore q should be clamped into the range $0-1$.

The point Q can be found by the following formula:

$Q = A + q \times \vec{AB}$

Now compute the distance d from Q to P and check if it is closer than r.

One final optimisation can be made here:

If $r < d$, then it follows that $r^2 < d^2$. So only need to compare square distances, which are faster to compute as no square root is required.

Algorithm 4: Local Path Following

Input: robot position R, position of end of local path E, maximum local look-ahead distance L_l, inflation map M, inflation radius I

Trace line from R to P_i against M

if *line doesn't intersect inflation* **then**

  $followMode \leftarrow GLOBAL$

$P \leftarrow$ Compute local path from R to E

$count \leftarrow -1$

$i \leftarrow 0$

for i **to** n **do**

  $d \leftarrow$ distance from R to P_i

  if $d > L_l$ **then**

    break

  Trace line from R to P_i against M

  if *line doesn't intersect inflation* **then**

    $localGoalPose = P_i$

    $count \leftarrow count + 1$

end

if $count = -1$ **then**

  // Local path is blocked!

  Clear and recalculate the whole costmap

  $c \leftarrow$ Nearest non-inflated cell at least $2I$ away from all local obstacles

  $localGoalPose \leftarrow c$

  if *length of* $P = 0$ **then**

    /* No local path exists, reset and switch to global path finding */

    $followMode = GLOBAL$

Move in a straight line towards localGoalPose

Fig. 5. Global Path (purple arrows) shown before dynamic obstacle. Green pointer shows the robot's position. Blue line is drawn to the current goal position.

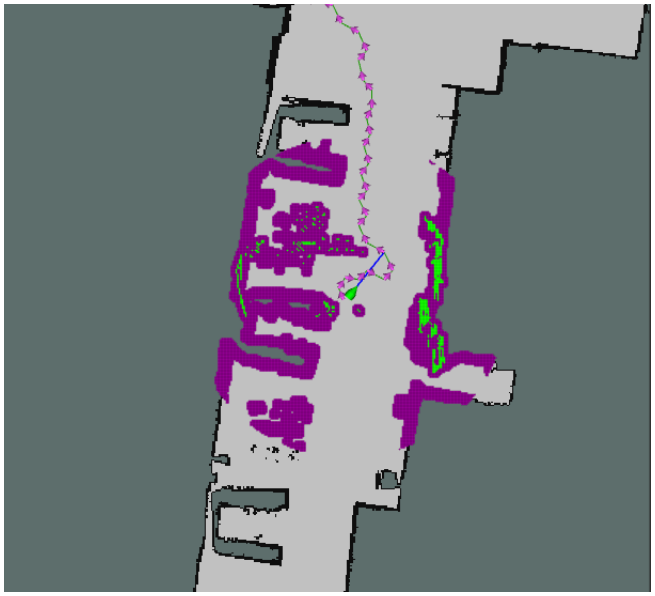

Fig. 6. Global Path (purple arrows) shown around dynamic obstacle. Green pointer shows the robot's position. Blue line is drawn to the current goal position.

these nodes, their 'F-score' (projected cost of path to goal) is incremented by a large constant value of 100 (found through trial-and-error). As the F-score is minimised by A* search this made them highly undesirable, ensuring they were discounted from the pathfinding except for in the most extreme circumstances.

A large constant was chosen to ensure instead of blocking these nodes entirely to ensure that a path is always found even if blocked by dynamic obstacles,

Figures 5 and 6 show the global obstacle avoidance visualised in RViz.

3) Local Obstacle Avoidance: When global obstacle avoidance fails, the robot switches to high-resolution local pathfinding. This involves maintaining a closeness field in the costmap around obstacles.

The local path is computed at a cell-level using Dijkstra's algorithm backed by a minimum priority queue (implemented as an array-based binary heap tree). A priority queue is used to ensure the algorithm is as efficient as possible.

The current position estimate is not completely accurate, therefore avoiding collisions is our top priority here, so the algorithm attempts to find a path with the lowest closeness. This means the path may not be the shortest in terms of distance, but is most likely to be the safest path as it tries to minimise the distance to obstacles.

4) Local linear Movement: This is the base motion that the robot performs. The robot simply turns to the goal, and once within a certain angle tolerance t drives forward.

The speed at which it drives forward is scaled down based on the proximity of detected obstacles being less than a danger threshold D.

Immediate collision checks are done, determining if (given it's current velocity) within the next second the robot will intersect with any of it's currently detected obstacles. If

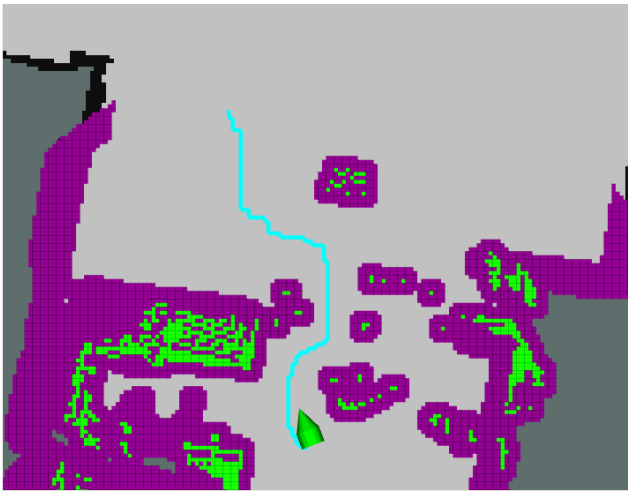

Fig. 7. Local Path cyan line shown around obstacles. Green pointer shows the robot's position.

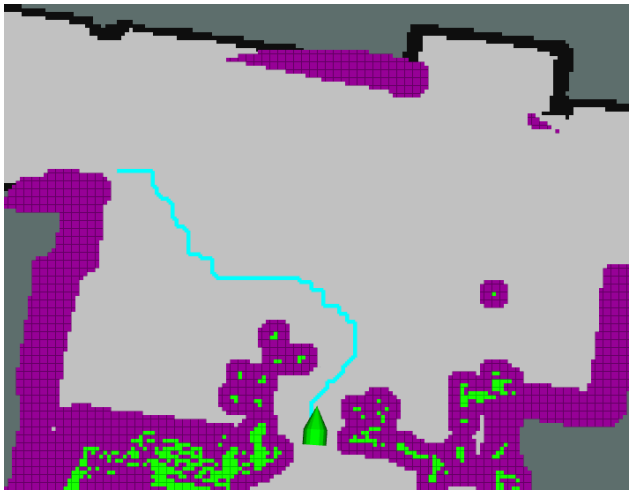

Fig. 8. Local Path cyan line shown after discovering more obstacles. Green pointer shows the robot's position.

Fig. 9. Actual layout of obstacles (stools) used for local obstacle avoidance test. Robot included for scale

so, the desired velocity is immediately zeroed to prevent collision.

t and D were set to 0.3 radians and 1 meter for the robot configuration.

F. Order Brain: Job Processing

1) Job queue/state: In order to identify what current job was being performed and give the robot a relevant goal, a node to hold the jobs in a queue with their current progress was created. This node can add and remove jobs, and it publishes on several topics:

- `/job/current` - The current job selected with its associated state.
- `/job/tables` - A list containing all the table locations.
- `/move_base_simple/goal` - A pose with the desired table location.

Some of the same topics used by existing ROS packages were chosen, notably `move\_base`, `amcl` and RViz to permit interoperability with differing localisation and motion systems. It also subscribes to a some fields in order for it to manipulate the job queue:

- `/amcl_pose` - Used to determine the current distance from the goal pose.
- `/job/add` - An interface for the phone app to use for adding jobs.
- `/job/advance` - Called to advance the jobs state, given the robot is close enough to the goal node.
- `/job/delete` - Removes a job from with the same ID given.
- `/qr_reader` - When the robot has arrived at the jobs table the final check is to see if the scanned QR code is equal to the job ID.

2) QR code scanning: Using a Python wrapper for ZBar [**?**] (the most prevalent QR code parser library) combined with OpenCV [**?**] (a simple image viewer/drawer) the QR node scans the webcam output for a QR code and then provides feedback to the user by highlighting the QR code outline and outputting the parsed code to the `/qr_reader` ROS topic.

3) Network: The communication of jobs/orders between the robot and the customers is handled by a Java server hosted on Amazon Web Services. The server permanently listens for new clients and the robot. It periodically broadcasts to all clients the current state of the robot where the state is whether the robot is connected or not and its pose if it is connected. Meanwhile a Python node on the robot permanently listens for new jobs from the server and broadcasts its pose to the server.

4) Customer Interface: A simple app was created for the 'customers' to input their orders using the interface shown in fig10. The app connects to the server hosted on Amazon Web Services. The user can see the status of the server and the robot (online/offline). If the robot and the server are online the user can input their name, select a table, and order one of the four drinks. Their order will be sent to the server where it will be forwarded to the robot. The user will then be presented with the second screen where they can see a live update of the robot's position and the QR code that they need to show the robot when it arrives.

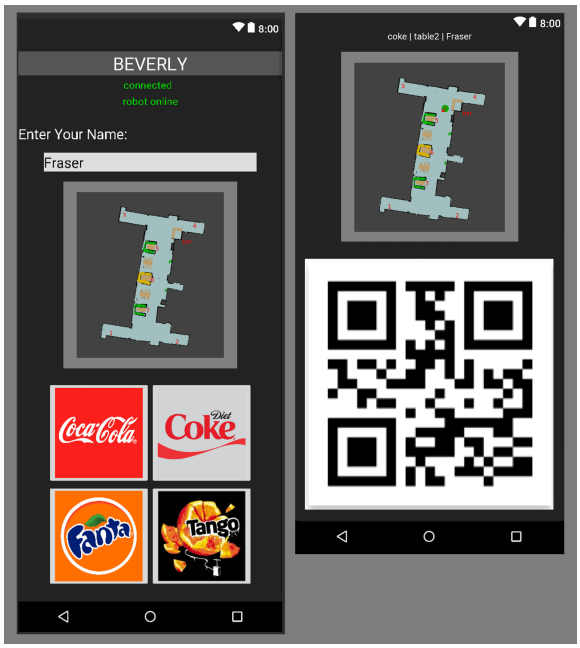

Fig. 10. The two screens of the customer interface implemented as an android app.

IV. EXPERIMENTAL RESULTS

A. Impact of node number in graph on pathfinding algorithm

1) Introduction and Hypothesis: Previously the pathfinding methods were discussed, specifically, the node maps used to give goals for the robot to travel to. In this experimental section, the trade-off of node count vs efficiency will be examined.

In order to investigate this, the pathfinding code was run to three different tables from the bar, and for each of these routes, four different graphs were used. The subsequent cost of the route, and number of nodes traversed was recorded. The path cost per node was then computed, and the average taken over this. The hypothesis was that the more nodes that were present, the cost and number of nodes traversed would be reduced as a more optimised path could be taken.

2) Results:

TABLE I

PATH COST VS. MAP NODE COUNT

Number of nodes	Goal table	Path cost (m)	Nodes traversed	Path cost per node (m/node)	Average path cost (m/node)
357	1	32.43124	63	0.514781651	0.491578
357	4	7.111131	14	0.507937929	
357	6	14.01242	31	0.452013452	
398	1	33.2205	67	0.495828313	0.51159
398	4	6.149115	11	0.559010455	
398	6	14.39512	30	0.479837367	
1428	1	33.79073	130	0.259928715	0.287281
1428	4	7.089021	20	0.35445105	
1428	6	14.35284	58	0.247462741	
5312	1	36.47805	260	0.140300208	0.169652
5312	4	9.990841	41	0.243679049	
5312	6	13.87238	111	0.124986396	

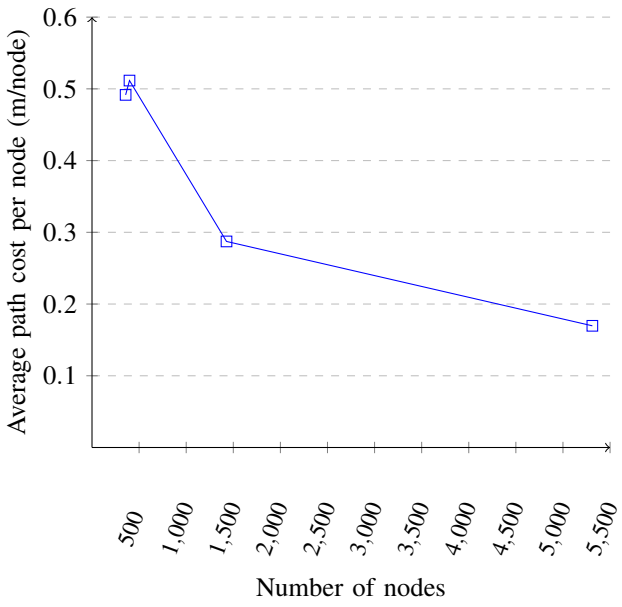

Fig. 11. Graph to show how path cost per node varies with number of nodes

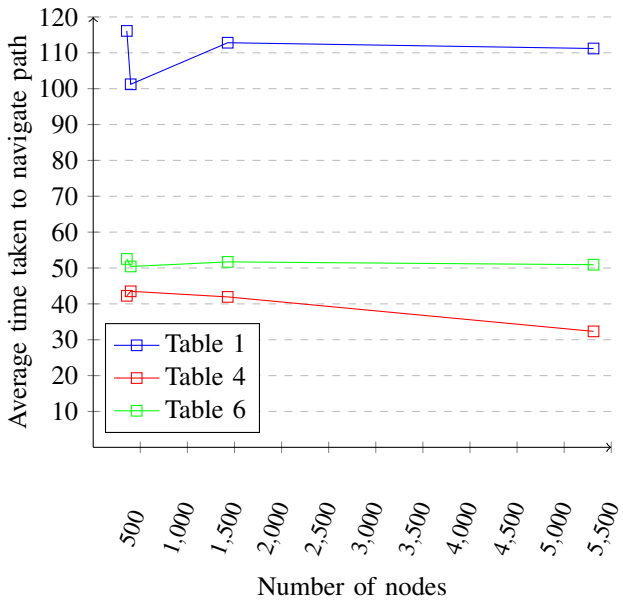

Fig. 12. Graph to show time taken to navigate path varies with number of nodes

3) Analysis and Conclusion: The results show a negative correlation between the number of nodes and path cos. However this doesn't hold true in the 5312 node graph where the routes to both table 1 and 4 are larger than any of the routes before. This is most likely due to a zigzag effect happening where due to the path-finding having to traverse too many nodes in an sub-optimal direction to reach the goal. This interaction between minimising path cost and zigzagging leads to an optimal node count around 1500 nodes.

B. Impact of node count on path execution time

1) Introduction and Hypothesis: In this experiment the actual time it took for the robot to arrive at goals with varying node count was examined. For 4 different node count the robot was given jobs that sent him to 3 different tables and the time was taken for each journey from the bar to the table. This was repeated 3 times to remove uncertainly. Our hypothesis was that this would mirror the previous experiment, with the time decreasing with node count increasing up to the 5500 node map where the time would increase again.

2) Results:

TABLE II

NUMBER OF NODES VS. TIME TAKEN TO NAVIGATE PATH

Number of nodes	Goal table	Time taken (s)			Average time taken (s)
		1	2	3	
357	1	116.68	117.23	114.38	116.10
357	4	41.62	43.45	41.69	42.25
357	6	52.79	52.32	52.45	52.52
398	1	101.41	102.23	100.01	101.22
398	4	44.32	43.41	42.78	43.50
398	6	50.16	50.11	51.01	50.43
1428	1	112.47	113.23	112.72	112.81
1428	4	41.97	42.13	41.73	41.94
1428	6	51.83	51.23	52.03	51.70
5312	1	111.27	110.34	111.98	111.20
5312	4	32.13	32.45	32.08	32.22
5312	6	50.91	51.04	50.82	50.92

3) Analysis: The results show that once the various optimisations that are detailed in the mover node are applied, node count has no visible effect on real time pathing. For each node count the time only varies by 10%. This is most likely due to the local pathfinding that is used to reduce path cost. If the robot can see a node that is later on in the path, it will skip out the previous nodes in favour of the shorted path. This nullifies the large quantities of node, and make the path in practices a few optimised points.

V. CONCLUSIONS

Experimental evidence shows that this robot waiter was successful. Systems were implemented, some using libraries, but most from scratch. These were integrated and managed to perform the desired results.

REFERENCES

[1] Ros.org. (2018). ROS.org — Powering the world's robots. [online] Available at: http://www.ros.org/ [Accessed 13 Dec. 2018].
[2] Wiki.ros.org. (2018). navigation - ROS Wiki. [online] Available at: http://wiki.ros.org/navigation [Accessed 13 Dec. 2018].
[3] Asif, M & Sabeel, M & Khan, Zeashan. (2015). Waiter Robot – Solution to Restaurant Automation.
[4] GitHub. (2018). ros-planning/navigation. [online] Available at: https://github.com/ros-planning/navigation [Accessed 13 Dec. 2018].
[5] ZBar. (2018). ZBar bar code reader. [online] Available at: http://zbar.sourceforge.net/ [Accessed 13 Dec. 2018].
[6] Opencv.org. (2018). OpenCV library. [online] Available at: https://opencv.org/ [Accessed 13 Dec. 2018].