COMP4034 – REPORT

Bradley Gallagher

Student ID: 20599466

1. ABSTRACT

Robots are often used instead of or alongside humans to perform repetitive, or difficult tasks. To successfully operate, robots need to follow sets of rules to meet its success criteria. However, for a robot to operate autonomously, its rules must be significantly more thorough as it cannot rely on human aid in the event of an error, and it should also be able to overcome uncertainties it encounters.

2. INTRODUCTION

For this task the robot (TurtleBot) should be able to autonomously explore the world, whilst avoiding obstacles, and searching for as well as remembering the location of specified objects. Due to the dynamic nature of autonomous robots, any behaviours developed should integrate with each other, and ideally be (re)usable with methods not developed within the scope of this project.

The primary behaviours for the robot will to be autonomous are obstacle avoidance to preserve the robot's (and its surrounding's) safety therefore satisfying Asimov's laws, object detection and pathfinding to the specified objects, and an exploration algorithm to explore the world in search of the desired objects.

In this report, the term "obstacle" will be used to refer to items where avoidance-based behaviours are required for self-preservation (e.g., walls) whilst "object" will refer to an item of interest which requires investigation (e.g., a red fire hydrant). However, objects can become obstacles under the correct criteria (threat of collision).

For the robot to succeed in its task, it would have to locate all (if any) objects of interest and announce the object has been found. It should also avoid driving over any blue floor tiles as well as traditional obstacle avoidance, it should mark obstacles and objects on the map once found, it needs to navigate the map in a way that maximises its search area, to localising within the map, and to utilise an appropriate control architecture to manage its behaviours to achieve its goals.

3. ROBOT FUNCTIONALITY

Asimov's laws are present throughout robotics, consequently behaviours such as obstacle avoidance are required to satisfy Asimov's third law for self-preservation, and possibly the first law if someone were to enter the robot's exploration area. Without any self-

preservation/safety response, autonomous robots would be unable to competently complete their tasks, therefore their ability ensure their safety is of paramount importance.

Before an autonomous robot can act or plan its next action, it must be able to sense its environment through its sensors. The simplest sensor to use for obstacle detection are active sensors, in this case lasers, as they also provide full coverage surrounding the robot (360 degrees) with a high refresh rate (30 scans per second) whereas an alternative such as the passive sensor of the camera (and its depth detection) only applies to the front of the robot, as well as usually having a slower refresh rate (dependent on the camera's hardware).

A higher refresh rate, as well as complete coverage is desirable so that nothing in the environment is missed. In addition, higher refresh rates are better at combating uncertainty by being able to compare data over time (or to another nearby sensor), therefore a singular erroneous measurement is less likely to cause errors. This can lead to a more consistent behavioural pattern, as well as a lower chance of the robot acting unexpectedly due to outdated or erroneous information resulting from a low refresh rate.

The usage of lasers will primarily concern the front of the robot whilst it is moving, however, it can also affect the rear of the robot whilst reversing, and in fringe cases the sides of the robot, as the TurtleBot cannot turn in place perfectly, therefore it requires adequate room to turn safely.

Alternatively, object detection utilises the robot's passive sensors, in this case it's camera, as it allows the usage of image segmentation to separate specific colour ranges from their original image. The primary issue of this approach is that colour masks are prone to (colour) noise, especially in real world scenarios. This is due to many objects sharing similar colours, as well as inconsistent lighting which alters the digitised RGB colour value of the object. This may result in the object no longer being within the desired colour range, or even the robot believing that an object is of interest when it is not.

Once an object of interest is spotted, the robot must move towards the object and remember its location in the world. In addition, there are blue floor tiles throughout the world that the robot must attempt to avoid driving over, however the task of identifying these objects is identical to that of any other colour, however the subsequent behaviour will be the opposite (avoidance instead of investigation).

Finally, the robot will require an exploration method to search the world for its goals. The ideal approach for this would be for the robot to remember where it has been before, as well as to identify locations where it has not yet been, however due to the complexity of the task, especially in larger locations, the ability to optimise this process is of extreme importance as the lack of the robot's degrees of freedom greatly limits the robot's ability to explore. Subsequently, anything left unoptimised could result in an inefficiency (i.e., a longer search time) that is magnitudes higher for larger areas of exploration.

4. METHOD DEVELOPMENT

Due to the module and project's structure, some of the behavioural functionality had been developed prior to starting development on mini-task 5. Consequently, the foundations of this functionality did not need to be re-developed, however it needed to be re-integrated, re-tested, and re-tuned so that it was compatible with any newly developed behaviours. The pre-developed behaviours were obstacle detection and avoidance, object detection (image segmentation) and investigation, an occupancy grid, and partial testing of the integration of ROS navigation stack (move base) functionality.

4.1. Control Architecture

Before starting development on any behaviours, a proposed solution and integration method for each of the specified requirements should be decided, as many behaviours could interfere with each other's solutions. Due to scope of the task, the following control architecture was used as a foundation due to its consistency with the hierarchical requirements of ROS programming.

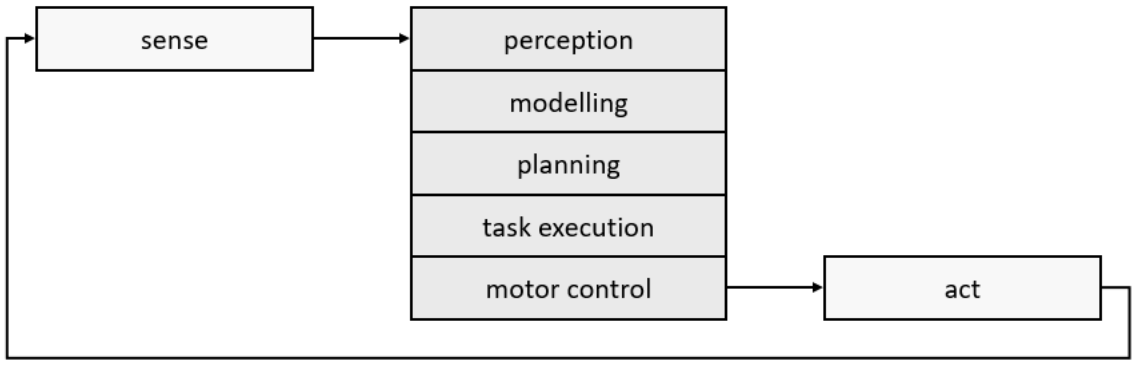

Figure 1. Sense-Plan-Act, hierarchical approach to behaviour control [1],[2]. Provides an outline of how autonomous robots should operate, and the orders of their behaviours.

A simplistic approach to behaviour control as shown in figure 1 can be used to extract each behaviour into a hierarchical order of operations. For this: "Sense" would be allowing the robots sensors to refresh (e.g., lasers, camera) whilst "Perception" would be updating any variables/values that rely on these sensors. By updating these variables, the process of "Modelling" would have been completed (e.g., updating an occupancy grid or detecting an objects/obstacles existence). Next, the "Planning" phase is significantly broader as this would include conflicting behaviours such as obstacle avoidance, object detection, and exploration. However, it would only need to designate one behaviour as being prioritised for "Task Execution" wherein any associated actions would be processed as part of "Motor Control" and subsequently "Act" being the execution of the behaviour. Finally, the robot would refresh its sensors repeat this in a closed loop until a success or failure criteria was met. In this case, an open loop would not be desirable as there is a success and failure criteria in fully exploring the world, and the robot endlessly exploring would not be efficient, nor desirable as an outcome.

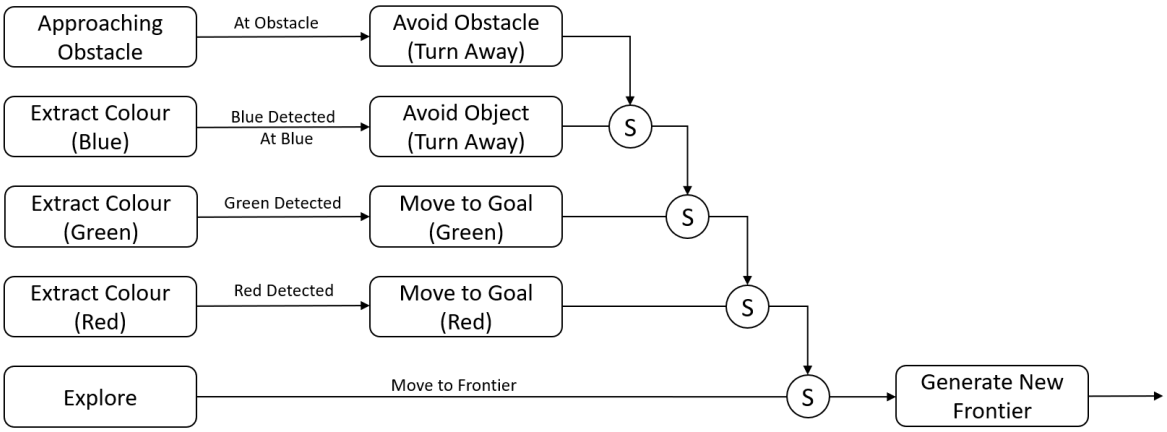

Figure 2. Subsumption Control Architecture for task execution/behaviour priority. Highest behaviours have priority, and when triggered they will suppress any behaviours below them. This graph is an extension of "Planning" from figure 1 by using priority as a planning mechanism.

A detailed approach to the "Planning" control mentioned in figure 1, is illustrated in figure 2 through subsumption architecture, as the usage of hierarchical layers are paramount for establishing behaviour priority in ROS programming due to its usage of nodes. Ideally, each category from figure 2, as well as unmentioned tasks for execution would be their own behavioural nodes (i.e., obstacle avoidance, object detection/investigation, occupancy grid maintenance/development, and exploration algorithm).

The control architecture of "Sense – Plan – Act" is relatively human-centric in its approach to problem solving through its visual stimuli, or lack of it. However, adjustments were needed to make it efficient as a robot, consequently by making it robot-centric through a greater reliance on lasers, odometry, and mapping, the robot would be more efficient at its exploration.

Most behaviour layers within the subsumption architecture are encapsulated by a closed loop. For this task, open loops were not desirable as most behaviours are conditional. For example, obstacle avoidance is not needed unless the criterion of a potential collision is detected. In contrast, through usage of subsumption architecture, the lowest priority behaviour acts as an open loop – while there is somewhere to explore (i.e. "True"), move towards the goal, unless a closed loop has met its criteria. This ensures that the robot will always have an active behaviour and should never fail to meet any behaviour conditions whilst operating. In addition, this allows for easier behaviour integration, as each respective behaviour will not produce any conflicts through suppression of behaviours lower in the hierarchy.

The highest priority behaviour is to avoid any obstacles detected within a distance that risks collision, which would satisfy Asimov's third law for self-preservation. The second highest priority behaviour is to avoid driving over blue tiles; however, this requires the blue tile to have been detected by the camera. The next two behaviours are for detecting green and red obstacles as well as investigating them, as the robot must find these objects safely to succeed in its task. These are separated into two behaviours to allow for objects to be identifiable by colour (distinct image segmentations), consequently by separating their

closed loop criteria by colour range detection, they become distinct behaviours which means one must be prioritised within the control architecture. Finally, when no other behaviour condition is satisfied, the robot should explore the world in search of its goals. Due to the relative simplicity of the task, the subsumption control architecture is relatively small, and does not need any inhibitors, as prioritising a behaviour through a hierarchy and using suppressors will satisfy the success conditions as well as ensure the robots safety.

4.2. Exploration Algorithms

Many types of robots find success through their imitation of human features, or behaviours, therefore by mimicking a human-centric approach to generate a solution, the robot would be more likely to succeed. Due to the task being identical to what humans do every day, an optimised behaviour already exists. In this case, when looking for an object, people will move into an area and visually search for the object. If they are unable to find the object, they will move to the next area. In this case, object manipulation is outside of the scope of this task. Although human-centric approaches are not always efficient for robots, it provides a good outline of what the ideal behaviours should be.

As an exploration algorithm is one of the most important behaviours for an autonomous robot, as well as the behaviour which will need the most testing, and tuning for any future behavioural integrations, its development was prioritised. Once an exploration algorithm was established, any additional functionality could then be integrated into it, which would minimise the inefficiency of developing unused solutions.

Although a wall following algorithm was previously developed, this alone is not an efficient approach to exploration as it cannot guarantee that the robot will move to a new (unexplored) area, it can also not adequately explore areas which lack walls. Consequently, an exploration algorithm would be required, as wall following alone would be insufficient.

The primary concern was the efficiency of the search algorithm used. For example, a landmark based navigation system may struggle to find any noteworthy landmarks in the simulation due to its dynamic nature. It would likely be unable to differentiate between the identical green, red, or blue objects, or be able to derive its location from them as they are not static objects. Although it may be able to use certain walls as landmarks within the simulation, this may not be reproduceable outside of the simulation, for the real-world demonstration, which leaves this approach as lacklustre in its potential for dynamic nature of the environment that the robot will be operating within.

An ideal goal for this task would be for the robot to map the area without needing an existing map, however this is not a feasible alternative initially, as it would complicate any future testing, as well as eliminate any usage of a global navigation strategy (and ROS navigation stack), as these require a map to function. Although an exploration algorithm could work through the sole usage of a local navigation strategy whilst building a map, this would lead to increased exploration times, or inefficient exploration which is contradictory to the desired goal.

The usage of a frontier exploration algorithm is ideal, as it would optimise the search process by identifying locations the robot has not previously or thoroughly explored, therefore maximising the search area by thoroughly searching every location within the map. Due to the uncertainty present in robot sensors and the nature of the task, a thorough search algorithm is extremely desirable to ensure that the robot does not miss anything of importance. This would require an occupancy grid to track the explored, and unexplored locations throughout the map, as well as a method to move to the new frontier.

A possible alternative to frontier exploration would be a PRM algorithm, as this may provide a more efficient pathfinding algorithm (providing a lower cost search, therefore being more time efficient) through its nodes and vertices. However, the primary problems with PRM's are that they struggle with enclosed areas, dynamic obstacles (placing an obstacle on an existing vertices/node), and cannot guarantee that they have explored the map thoroughly. Consequently, the application of a frontier search algorithm should be more consistent in its search at the expense of a more expensive search.

4.3. Mapping

To utilise a frontier search algorithm, an underlying map must exist to provide the locations of occupied, free, or unknown areas. For this task, a semantic map would overcomplicate any approaches or solution implementations, therefore it would not be an ideal solution. The use of a topological map would provide an effective method for exploration, however by providing connected nodes and edges, the robot may be unable to thoroughly explore areas where a node does not exist. In addition, topological maps lack scale which means that an autonomous search algorithm may not be efficient at calculating the closest node whilst considering obstacles between the robot and its goal. Finally, a discrete metric map through an occupancy grid would allow for the map to be sufficiently explored, as well as allowing for frontiers to be at any possible location throughout the map. Consequently, a metric map would also provide additional adaptability which would allow for further tuning and testing which could result in a more efficient solution.

Before anything could be developed for the occupancy grid, a size and resolution would need to be decided. I used a size of 41x41 grid with a resolution of 0.5 metres per pixel. This size allowed for the simulation map to always fit onto the grid with the odometry position of 0, 0 equalling [20][20] within the 2D array. Consequently, any map position could be designated as 0,0 and the occupancy grid would still work. This would need to be tuned for different maps/map sizes, by changing the centre position, as well as the size of the grid, and if desired, the resolution.

The usage of a high resolution was to simplify the occupancy grid so that there were fewer locations to verify during manual spot checks and troubleshooting of any integrated behaviours, this also resulted in fewer possible frontiers existing which greatly simplified the exploration process and led to less downtime while selecting a frontier. In addition, by potential frontier positions being a larger resolution, a buffer was created around obstacles which mimicked a configuration space, as if an obstacle was detected, the closest the frontier could be is 1 pixel away (illustrated in figure 3). This greatly improved the chances of

the robot arriving successfully at its destination, as it would not struggle with realigning itself alongside an obstacle to achieve its goal. However, while testing an occupancy grid with a low resolution (0.05), it was found that the robot would struggle to detect neighbouring pixels through its usage lasers. The robot would also try to move 5cm at a time due to the frontier algorithm detecting an unknown neighbouring pixel, therefore the resolution was increased as this had provided superior results historically.

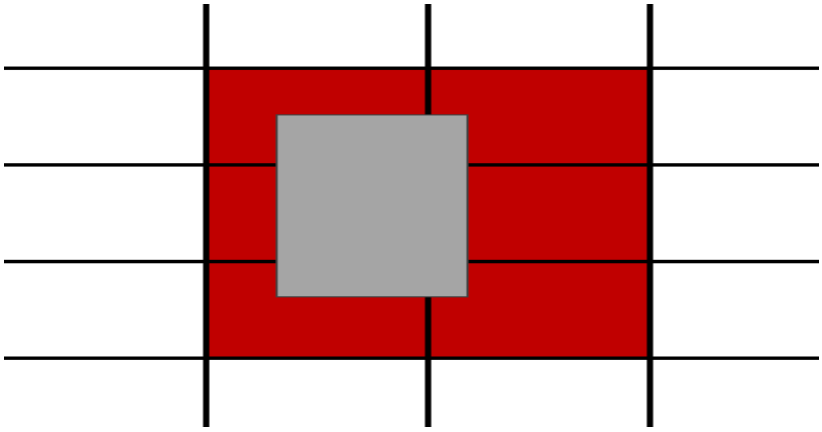

Figure 3. An illustration of an occupancy grid – a grey obstacle, and red squares indicating occupancy pixels that are designated as occupied and cannot serve as frontiers for exploration. This means that no frontier goal can be less than 1 pixel away from a detected obstacle.

With the occupancy grid created, the robot would be able to use the use its lasers to determine the distance from the robot to the object/obstacle and add that object/obstacle into the occupancy grid and using the provided “to_line” function and add any empty space in between the robot and obstacle into the grid as well. However, before this could be integrated, the “to_line” function would need to know both the starting location and the end location of the laser.

Since the starting location was the robot’s current location, the odometry position could simply be converted into an occupancy grid pixel. However, the location of the lasers end position was not as simple. Since the robot only knew the direction of the laser, and the distance between the robot and object, trigonometry would have to be used to calculate the odometry position, as just adding the distance to a cardinal direction would have a large margin of error.

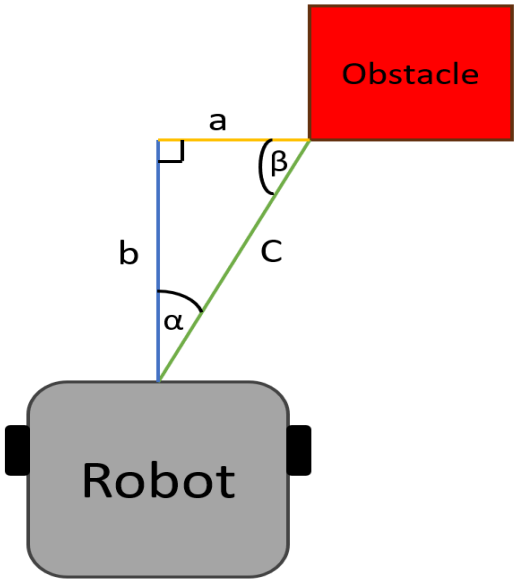

Figure 4. Trigonometry example for determining location of an obstacle through laser information. C represents the laser distance, and α represents the angle of the laser used.

Using figure 4 as an example for the solution implemented, since any laser angle used can be connected to its closest cardinal angle (0, 90, 180, or 270 degrees), a right-angle triangle can be formed with the closest cardinal direction to use the following calculations.

By knowing alpha (α) as the laser's angle, and the laser distance (c), we can determine both the length of a and b with the following formulas: a = c x sin(α), b = c x cos(α).

An additional problem is that the information being logged was in position relative to the robot, not the map. Consequently, the robot's lasers would always update a specific direction in the occupancy grid, regardless of the robot's theta direction which resulted in the occupancy grid being unusable due to its inaccuracy. This was due to the robot believing that its front laser was always North, instead of being influenced by the robot's theta.

This problem was solved by determining the robot's current direction from its theta, and then offsetting each laser angle by the current theta. For example, the left laser would be recalculated as: theta (as an angle) + 90 degrees, and if the sum of these is greater than or equal to 360, the result would have 360 subtracted from it to ensure that it remains consistent with the fundamental rules of angles.

In addition, by knowing direction of the laser, the odometry position can be derived by implementing the variance of a and b onto either the X or Y odometry axis depending on the laser position (a and b are not always the X and Y axis respectively). This allowed for the position of each object/obstacle to be plotted into the grid accurately, regardless of the robot's orientation.

This approach was tested by preplacing the robot into various positions and orientations and printing the results (the occupancy pixel which was updated as either occupied or free). Each laser was then compared with each other to confirm that they maintain directional

consistency both within the current scenario, as well as future scenarios with differing orientations.

4.4. Navigation

Once the occupancy grid was operational, the process of developing the frontier exploration algorithm began. The first goal was to find all possible frontiers – an unoccupied pixel within the occupancy grid which borders at least one unknown pixel. With the list of potential frontiers, the distance from the robot to each frontier would be calculated, and the closest frontier would be returned as the next goal/frontier for exploration.

A known pixel was used as the frontier, as attempting to move into an unknown location would regularly result in the destination having a fatal cost (i.e., the destination was inside of a wall), therefore using a safe unoccupied location should result in each frontier being achievable. By arriving at the frontier, the unknown neighbouring pixel should then be explored by the robot's lasers.

During testing, the frontier exploration algorithm encountered the issue of having a null value wherein no frontiers exist as it is not possible to explore any further. This was resolved by adding a new closed loop which would shut down the ROS node once exploration was completed, as if the map can no longer be explored, it is unlikely that the robot will be able to find any new objects of interest. Alternatively, if the map was fully explored, it would be meaningless to continue to explore the world, as no new object should be detected in any future iterations if the first iteration had thoroughly explored the world.

With the new frontier goal provided, the robot would need a way to navigate towards that goal, and the previously developed wall following algorithm would not be able to achieve this. Consequently, the existing ROS navigation stack (move base) was used, where the occupancy grid frontier would be converted into an odometry position and provided as a goal to the move base client. Only one goal was provided at a time to the client, as providing multiple goals (forming a queue) would result in the inefficiency of the next frontier being at the start location instead of its new and current location. This back-and-forth exploration is undesirable, therefore only one destination existed at any given time.

The primary issue encountered with move base is that the node hosting the client is unable to multitask move base and other behaviours simultaneously, as doing so would result in the robot becoming unresponsive to either the move base client, or looped behaviours, and would collide with obstacles as it was unable to receive or process any evasive self-preservation behaviours. Consequently, processes such as updating the occupancy grid or investigating objects would have to be processed whilst move base was not operating.

The solution found for this was to allow move base client to operate for one second uninterrupted, to then cancel any move base actions whilst other processes were underway. This resulted in negligible stuttering within the robot's movements, however these were not explicitly noticeable as any other processes would either be complete before the robot had decelerated, or the alternative processes would become the priority behaviour instead. This was an efficient solution as cancelling move base does not suddenly decrease the robot's

speed to 0 m/s, therefore the robot would only suffer minor deceleration whilst processing additional functions, only to then return to its original speed.

The time in which move base was allowed to operate was tested to allow for two or more seconds, however this resulted in many issues as all other behaviours would be unresponsive for this time. As a result, objects of interest were not spotted, as the parameters the behaviours relied on were never able to be updated. Consequently, a minimalistic period of one second was ideal for this solution.

The usage of move base was mostly consistent and reproducible throughout its testing, however it would struggle with uncertainty while trying to plan a path to an unattainable destination which would render the robot stationary for a period until the goal was automatically cancelled by the client. However, this would not change the frontier goal, therefore the unattainable goal would be repeatedly provided to the move base client without any action being taken as the goal was still unattainable.

This was resolved by implementing an odometry distance tracker to track how far the robot had moved in the past one second of move base operating – if the robot had not moved a noticeable amount, it is unlikely that an achievable path exists, therefore the frontier should be temporarily discarded and a new frontier generated. Any frontier that was unobtainable by move base was temporarily changed to 2 within the occupancy grid to designate that it is unoccupied, but currently ineligible as a frontier. Consequently, using known pixels in the occupancy grid as frontiers would allow for easier manipulation of the grid without having to assume the status of an unknown location.

Although odometry suffers from uncertainty, such as dead reckoning, by using short intervals for the distance tracker, the uncertainty would be negligible or non-existent, therefore having no impact on any behaviour reliant on this measurement.

The final problem encountered within the usage of move base is that the robot would occasionally reverse either towards its goal, or to realign the robot after avoiding an obstacle. Although this is beneficial in obstacle avoidance and path planning, this did present an issue in the avoidance of blue floor tiles, as if the robot's camera (on the front of the robot) is unable to see the object (behind the robot), no evasive behaviour can be enacted.

The possibility of disabling reversing was investigated, however this generated new issues wherein the robot could not manoeuvre in tight areas as it would require larger turning circles which severely hindered move base's ability to generate plans. Consequently, the robot must be able to reverse when needed, however its usage should be limited to where necessary.

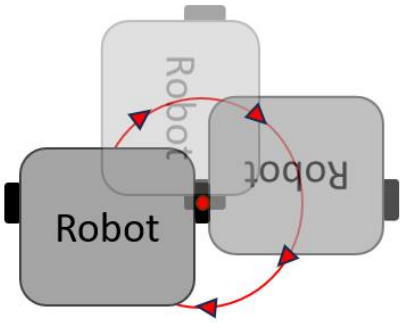

Figure 5. The robot's centre of rotation and stationary turning circle if its ability to reverse is disabled. The centre of the turning circle is represented by a red dot in the centre of the right wheel (for right turns, or the left wheel for left turns), and surrounded by a red circle which indicates the path the robot will follow while turning.

An example of the increased manoeuvrability needed by the robot after disabling reversing can be seen in figure 5. If the robot is no longer able to reverse whilst attempting to turn whilst stationary, it may only use one wheel to move which drastically widens the turning circle. Although this issue primarily affects the robot whilst it is still, if the robot becomes stuck, it will be unable to reverse out of the situation.

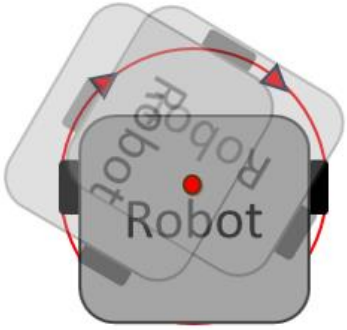

Figure 6. The robot's rotation centre and stationary turning circle if it can reverse. The centre of the turning circle is represented by a red dot in between the robot's wheels, and surrounded by a red circle which indicates the path the robot's wheels will follow while turning.

To emphasise the difference in space needed between the two, figure 6 demonstrates the reduced turning circle of a TurtleBot which can reverse one of its wheels whilst turning. Please note that due to the forward position of the robot's wheels, it is unable to turn in a perfect circle, and the rear of the robot will protrude whilst turning, but to a significantly lesser extent to that illustrated in figure 5.

The solution used was for when the robot was reversing (detected by using the theta and the difference in direction of the current odometry position from previous odometry position prior to move base's one second operation) the move base goal should temporarily be cancelled, to then be resumed after a short (1s) pause. This would usually result in the robot driving forwards towards its goal, however in cases where the robot could not move forwards, or doing so would be inefficient for its pathfinding as the goal was behind the robot, the robot would continue to drive in reverse.

Since the robot's ability to track its directional movement is limited by the odometry axis, there is minor uncertainty in this function regarding the robot's movement direction at slow speeds. This drawback was not spotted during testing as move base does not use speeds that could trigger such uncertainty.

Although the effects of reversing had been slightly diminished through the implementation of this solution, it was still able to reverse indefinitely, with short stutters. The solution was then adapted to keep track of how many times in a row the robot had reversed successively, once it had done so 3 times in a row, the robot would be forced to complete a half-turn to face the direction it is currently attempting to drive in. This largely resolved the issue, however if the robot was surrounded by blue tiles, it may be forced to turn away from the blue tiles only for move base to instruct it to drive over them in reverse – where the avoidance behaviour cannot be triggered.

From my testing, the reduction of reversing does mitigate the risk of driving over blue tiles, however it would not be possible for the robot to consistently avoid blue floor tiles without updating the cost map to prevent a path from overlapping with the tiles position.

Alternative tuning approaches were tested for this. Namely allowing for greater or fewer successive iterations of reversing. It was found that allowing for more iterations only allowed the robot to continuously and erroneously move in reverse which was undesirable. Meanwhile allowing fewer iterations would constantly turn the robot around, only for the goal to now be behind the robot, therefore it would once again need to turn around. Consequently, a middle ground was used in 3 as it allowed some leeway of error, without punishing uncertainty.

4.5. Object Detection

With the robot now able to dynamically explore the world, it needs to be able to detect, investigate, and remember the locations of any distinct objects of interest that it encounters. The first part of this problem is being able to detect these objects visually. Although object recognition would be ideal for this task, it is an overcomplicated solution to the problem. In addition, object recognition may struggle to detect blue floor tiles as they only stand out due to their colour. Consequently, due to the distinct colours of the objects of interest, the usage of image segmentation and colour slicing is both the simplest, as well as likely being the optimal solution relative to the task.

The easiest way to detect colours is to use image segmentation wherein only colours within a specified RGB range will be copied into a new variable which stores the new images data.

However, due to the sensitive nature of colour detection, and how prone to noise it is, the image was first converted from RGB colour values to HSV values as this will make the colour (range) detection less sensitive to lighting changes. Although lighting is not an explicit problem in a simulation, it is optimal to use an adaptive solution which can be used in the real world if needed, as the real world will suffer severely from lighting changes.

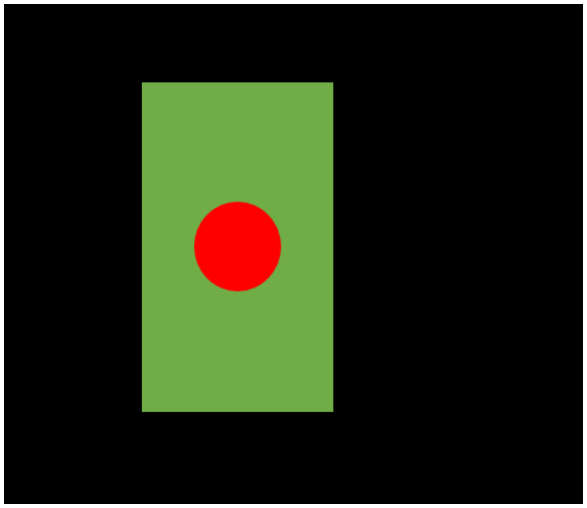

Figure 7. Illustration of a green colour mask, with a green object and a centroid (red dot) – the robot would want this red dot to be central whilst it is moving towards the object.

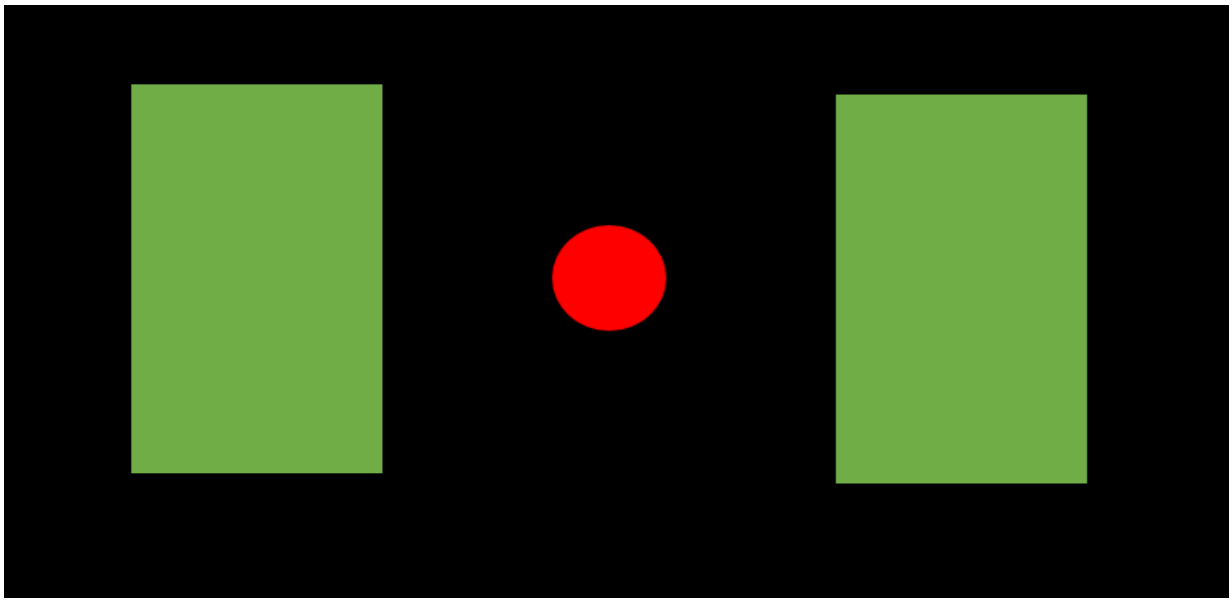

Figure 8. Illustration of a green colour mask, with 2 green objects and a centroid (red dot). The robot wants the red dot to be central whilst moving towards the object however, this will not currently take the robot to either object.

Once a colour had been sliced from the original image, the usage of contours (edge detection) can be used to detect where colours have changed to the desired (sliced) colour. With this information, a centroid of the contours can be calculated (seen in figure 7). The primary problem of this approach is that only one centroid can exist which means that if multiple objects are detected (as can be seen in figure 8), the robot's behaviour would only act as if one object at the location of the centroid instead of either object's true locations.

However, this problem is mitigated in practice as the closer the robot gets to the objects, one object will become the larger object which means that the centroid will slowly drift towards a singular object (as can be seen in figure 9).

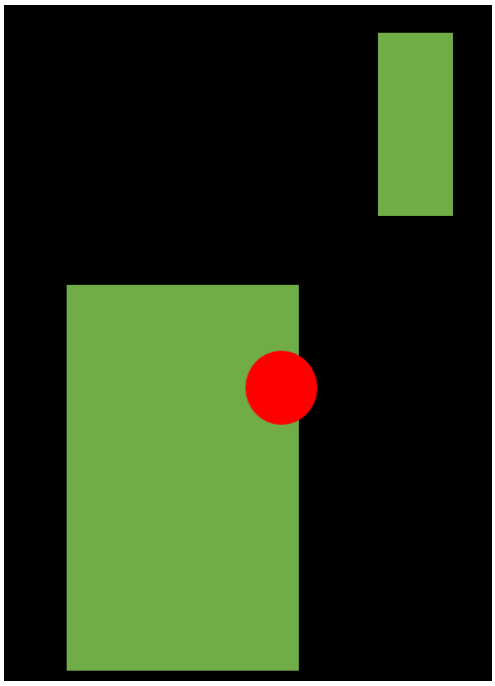

Figure 9. Illustration of a green colour mask with 2 green objects at varying distances and a centroid (red dot). Although the robot wants the dot to be central, the closer it gets to the left (closest) object, the more bias the centroid will have to the left (closest object).

With the colour mask's centroid generated, its X and Y axis position within the image can be exported. However, if no colour was detected, a default centroid location would be assigned of image height or width divided by 2 (this is created due to an exception for dividing by 0). Whenever the centroid was not at its default location, action should be taken to avoid the blue object, or investigate the green object.

Although it is possible for the centroid to be in the default position for a legitimate reason, this uncertainty is counteracted by the usage of float values making it extremely unlikely to be an exact match with the integer position of the default position. In addition, any movement the robot makes would offset this balance, resulting in the object once again being detected.

Whenever a green or red object was detected, the centroid's X axis position would be used to control the robots turning speed (published through twist) so that the robot would continuously aim at the object of interest. The robot's speed was also set to 0.2 m/s so that the robot would move towards the object, whilst the centroid would maintain the correct direction. This resulted in the robot repeatedly crashing into the object as no clause was in place to prevent the robot from crashing into the object it wanted to move towards. In addition, this behaviour was tested in isolation from other behaviours, which meant that the obstacle avoidance behaviour was not integrated during this testing process.

To prevent any collisions during this behaviour, the robot was limited to only being allowed to move (during object investigation) when its front laser distance was greater than 0.5 metres or was null (no object was detected within the maximum laser range). However, this in turn created the issue of wheel drift upon sudden braking when the laser distance became less than 0.5 metres. Consequently, the robot may no longer be aimed in the correct direction (due to carried momentum and wheel drift) to correctly log the object's location, therefore when the robot was within 0.75m of the object (determined by the front laser distance), the robot would decelerate to a speed of 0.1 m/s so that when it came to a sudden stop, it would suffer minimal wheel drift due to its lesser momentum.

It was found that by combining green and red objects into a single mask, due to their identical behaviours, differentiating objects of differing colours would be impossible as the behaviour was consistent for both cases. Therefore, it was decided that each colour should have their own separate mask and behaviour to correctly categorise the colour of the object when interacting with it. These behaviours were identical excluding their colour range for image segmentation.

Once the robot was within 0.5 metres of the object and had stopped, it would determine the odometry position of the object (see figure 4 for the calculation used) and update the occupancy grid to reflect that the obstacle in question is an object of interest, which should prevent the pixel value from changing, as well as allowing the system to recall the objects position. The occupancy grid used a value of 3 for green objects, and 4 for red objects. This was done to differentiate the objects from obstacles as well as provide readability in case a manual check of the occupancy grid was required.

In addition, the object would also be added into a list which stores all the objects of that type which have been discovered, and a command line message would be printed which states the number of the object found (of that type) as well as its odometry location and occupancy grid position.

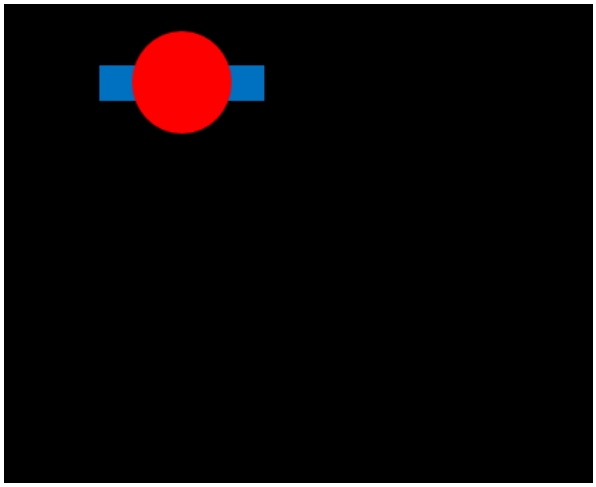

Figure 10. Illustration of a blue colour mask with a distant blue tile and a centroid (red dot). The robot does not want to have the red dot outside of its default position – it will attempt to turn away to avoid detecting any blue tiles.

In contrast, the detection of blue objects required a different behaviour upon detection since the goal was to avoid them rather than locate them. Consequently, the centroid for the blue mask was used to turn the robot away from the blue tile. This resulted in the robot being continuously interrupted so that it can turn away from blue objects detected in the distance which made exploration impossible due to constant unnecessary interruptions. An example of this undesirable scenario is illustrated in figure 10 where the robot is not under risk of driving over the tile, but still seeks to avoid it. This resulted in the behaviour to avoid blue obstacles only being fired once the blue object was close to the robot (seen in figure 11) – this was determined by the Y axis position of the centroid.

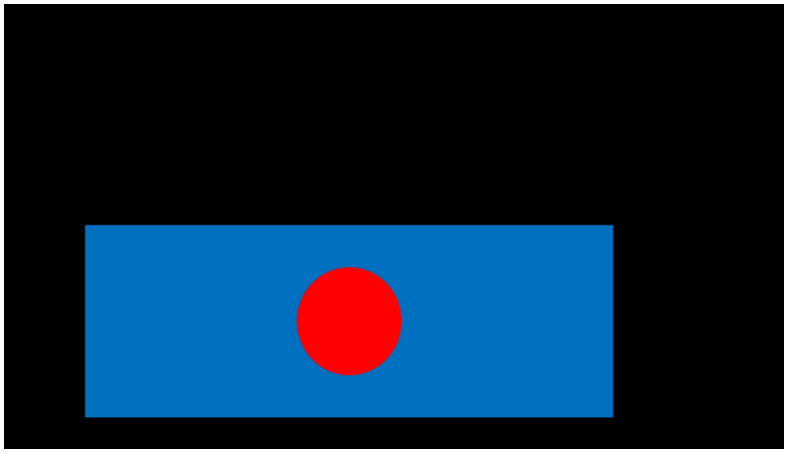

Figure 11. Illustration of a blue colour mask with a nearby blue tile and a centroid (red dot). The robot does not want to have the red dot outside of its default position – it will attempt to turn away from the blue tile now that it is close enough for the behaviour to fire.

This solution appeared to provide a consistent solution when facing blue tiles, however two major problems arose from this solution. Firstly, if the blue was detected nearby in the corner of the camera's vision, the robot would want to turn away even though it would not come into contact on its current trajectory. This was resolved by further tuning the firing condition to only occur if the centroid was within a specified X and Y range. Secondly, if the centroid was perfectly central, the robot would struggle to turn away from the blue object as it could not decide which direction to turn. This required a primitive solution wherein if the centroid was within a central range, the robot must be forcibly turned around until no blue object is detected (within the firing range) on the camera. Without this, the robot would become stuck in the blue object avoidance behaviour, however the centroid position could not be converted into a turn speed for the robot to escape therefore leaving the robot trapped.

The testing and tuning for blue object detection worked on individual blue tiles, however when the robot was placed within a cluster of blue tiles, the robot would become stuck continuously turning to and from different tiles, being unable to escape this loop. This problem also required a primitive solution where if the robot processed too many blue avoidance behaviours in succession, the robot should be forcibly turned around until no blue objects exist within the firing range. This largely solved the issue, but at the risk of the new

frontier path forcing the robot into reversing over the blue objects it had just attempted to avoid.

These three behaviours were integrated into the frontier exploration algorithm as demonstrated in figure 2. Whenever a colour object met its firing conditions, move base would become inactive, and the robot would be fully controlled by the fired behaviour until its success or failure was determined. However, it was found that the robot had no way of knowing if an object had already been found, therefore if it spotted one, it would always want to investigate it.

Consequently, the solution was to determine the robot's current theta direction and check the occupancy grid for if an object already exists in that direction. If so, the robot would not need to investigate as it had likely already done so, otherwise the investigation of the object would be warranted. This resulted in a minor inefficiency where if two objects were in line (like figures 8 and 9) and only one had been logged, the robot would treat both as detected, until the logged object was no longer in the robot's current direction. This was usually mitigated by the robot's thorough exploration so that it approached each object from various angles.

4.6. Obstacle Avoidance

The final behaviour to integrate was for obstacle avoidance, however the introduction of move base already introduced this to some extent. Consequently, the application of any obstacle avoidance behaviour would largely be relegated to any behaviours that were manually created and took control away from move base. This was mostly limited to ensuring that there was ample room surrounding the robot prior to turning to ensure that the robot would not collide with an obstacle whilst attempting to turn. This is best performed within a closed loop where the behaviour only occurs if the robot is at risk of collision. Examples of the robots turning circles can be seen in figures 5, and 6 which demonstrate that the robot is unable to turn in a perfect circle, as the rear of the robot will protrude whilst turning, therefore it must ensure that enough room exists before any manoeuvre.

However, the notable exception to this is that move base can occasionally drive closely to walls wherein the robot would be unable to turn if it became stationary. This should ideally be avoided and should be considered when designing an obstacle avoidance behaviour. As a result, the previously developed obstacle avoidance behaviour was integrated where if any obstacle was detected in front of the robot less than 0.3 metres away from an obstacle, the robot would stop and turn left until there was no longer an obstacle in front of the robot.

This reduced how close the move base path would bring the robot to any walls, however it would greatly slow down any attempts to reach a destination parallel to a wall, however this had already been mitigated through the configurate space created by the larger occupancy grid resolution.

This behaviour was attempted with greater and lesser ranges, however larger ranges proved too sensitive without providing a beneficial solution to the problem. Alternatively, a smaller

detection range failed to properly fire when close to walls which meant that the behaviour would never be triggered as move base would trigger its own obstacle avoidance before this behaviour could be invoked, but if move base was cancelled, the robot would be too close to the wall to safely turn.

5. METHOD DISCUSSION

5.1. Cost Map Manipulation

By solely relying on the camera to avoid blue tiles, as well as only having a front-facing camera, the robot can only avoid the obstacle that it can see. Consequently, the robot struggles to avoid these obstacles when near one or more, especially when they are out of the camera's field of vision. Since this problem is primarily triggered by move base, the optimal solution would be to integrate functionality which allows the cost map to designate blue regions as having a fatal cost. This would result in move base's path planning avoiding these regions. Although this would not completely erase the threat of error, it would significantly reduce the possibility, but this would require behavioural changes to adequately log blue objects into the cost map. However, this solution would likely reuse the existing investigation behaviour for red and green objects.

An additional optimisation through the manipulation of the cost map would be to increase the cost of moving near walls or objects as it was commonly found that the robot would drive closely to walls, which would prevent the robot from being able to turn without a collision. In addition, the robot may attempt to move into a tight area, specifically nearby the objects of interest, as the path planner did not recognise the objects existence prior publishing the plan, as they do not exist in the initial version of the map. As a result, the robot would become stuck between obstacles, unable to turn, or move forwards or backwards without entering a proximity to an object which would cause a collision.

5.2. ROS Navigation Stack Dependencies

On the other hand, much of the final version's functionality was reliant on RVIZ's Monte Carlo localisation, and pathfinding which meant that if the robot was required to operate within an uncertain, unexplored/unmapped environment, it may struggle to avoid problems such as dead reckoning, localisation, or produce an efficient pathfinding solution. Consequently, any desire to not use an existing map would require the development of a new pathfinding algorithm based on the occupancy grid (or alternative map), as well as a self-localisation method to mitigate dead reckoning from manipulating any odometry readings, consequently resulting in an erroneous and inconsistent occupancy grid which would only accelerate any errors in the robot's behaviours which are dependent on its accuracy.

Alternatively, when a new frontier is selected, it is done so only if it borders an unexplored area. Even if the robot is nearby, it may not be able to detect the unexplored area which

means that the frontier may persist into the future, as its surroundings were not logged thoroughly. This problem is also worsened by the occupancy grid only being able to update whilst move base is not operating, which results in minor durations of downtime wherein the occupancy grid is not being updated. This means that instead of utilising the laser's refresh rate of 30 per second, the lasers are only being utilised once per second. This was mitigated through the usage of a larger grid resolution; however, this does not solve the problem, consequently this could be resolved by implementing a probabilistic occupancy grid which would be able to estimate the occupancy status of neighbouring pixels within the grid which were not detected either due to a blind spot within the lasers, or during the downtime in between the occupancy grids update/refresh time.

5.3. ROS Programming Practice

Unfortunately, for the accessibility of the code, it does not follow standardised best practices for ROS programming wherein one behaviour is encapsulated per node. Consequently, nodes cannot be easily reused, instead methods or classes must be exported into the new project and then tuned.

Subsequently, an optimisation for this would be to restructure the codebase to follow best practices by storing each behaviour as its own node. This would also require the adjustment of behaviour suppression and loop firing parameters to ensure that behaviours do not conflict with each other.

5.4. Image Segmentation Noise and Uncertainty

Whilst the robot is searching for objects, uses edge detection within its red or green masks. This results in extreme sensitivity to any noise detected by the camera, as even one pixel of noise can alter the robot's behaviour. Since colour detection through cameras are prone to noise, especially in the real world, the ideal solution would be to only act upon edges which are of a certain size or larger. This would largely reduce the issue in smaller areas; however, the robot would struggle to detect distant objects as their edge size would no longer be large enough. However, this distance this would require would be outside of the scope for this problem.

In addition, the current method for detecting the distance to blue objects is determined by the Y axis position of the blue mask's centroid. This approach is extremely sensitive to errors as a distant object would be able to easily manipulate the centroid, in addition the Y axis position is not a real or consistent measurement as it is dependent on image size. Therefore, an ideal alternative would be to use the centroids position within the image and convert that position into an identical image which tracks depth instead of colour. The usage of image depth would allow for a more consistent measurement to be taken, therefore allowing the robot to make better informed actions which are not as affected by noise, uncertainty, or even image resizing.

5.5. Directional Measurement Optimisation

Many of the current occupancy grid methods are currently very primitive and rely on the usage of cardinal directions rather than angles. The primary victim of this approach is the

“ignore” function which prevents the robot from investigating an object, if an object has already been logged in that direction (as seen in figures 8 and 9). An ideal optimisation for this approach would be to both integrate the “to_line” functionality to search the occupancy grid within the cameras field of view rather than a cardinal direction, as the accuracy of cardinal direction-based checks exponentially deteriorates over distance. In addition, the integration of a camera depth measurement would allow a distance limit to be integrated to this check, as if two objects were in a line, they would both be ignored.

5.6. Exploration Algorithms

Finally, the usage of occupancy grids has provided a thorough search of the world, however such a thorough search also proved to be inefficient for timely searches. Consequently, if the world has already been mapped, it would be more efficient to use a topological map, as travelling to a node, only to turn around to visually search for objects would be significantly faster. Although this search would not be as thorough, very few maps would require such a thorough search, whereas the time efficient solution a topological map would provide is ideal for any task relating to object location. However, the primary downside would be that the creation of a topological map would likely need human intervention which brings doubt over how autonomous the robot would truly be.

Alternatively, a hybrid between a PRM and frontier search algorithm may prove promising. This would both provide lower cost (faster) exploration, whilst allowing the frontier algorithm to thoroughly explore areas surrounding the nodes. However, since this was not tested, the efficiency of such an approach is purely speculative.

6. CONCLUSION

The usage of subsumption control architecture proved to be effective in its behaviour management through suppression as well as its ability to aid planning of behaviour integration. In addition, it allowed for behaviours to be developed and tested in isolation from each other which resulted in a mostly seamless integration of behaviours, as only a condition was required for the already developed closed loop to run. If there was a problem with the behaviour, it would only need to be tuned to better respond to achieve the desired outcome.

However, due to the complicated nature of autonomous robots, there is abundant potential for the operation of these behaviours to be optimised, however this would depend on the desired outcome. The primary subject of this is speculation regarding the efficiency and performance of alternative exploration algorithms including PRM, topological maps, or a hybrid map. These methods may allow for swifter map exploration, at the expense of how thorough their searches are, however due to the nature of this task and the speed of the TurtleBot, only performing one search would be optimal. Consequently, the usage of a metric map with frontier exploration was optimal for performing a thorough search.

The primary issue present within the current model is that the robot is unable to update the cost map used by the ROS navigation stack. Therefore, it struggles to consistently meet the objective of avoiding blue tiles. However, it successfully and consistently satisfies all other criteria throughout its operations.

Consequently, the current model is capable of autonomously exploring in search of objects, as well as returning the position of those objects. In addition, the behaviours seek to optimise the behaviours by refining the firing conditions of behaviours to ensure that actions are only taken when they are needed, as seen with the "ignore" functionality. It achieves this while satisfying Asimov's laws where applicable. Finally, all behaviours developed in this project may be reused for other TurtleBot projects.

7. MISCELLANEOUS

Unfortunately, since throughout the project there were no functional behaviours submitted by my lab-partner, I was unable to test or integrate any code with my team-mate.

8. REFERENCES

[1] L. Felton, "Autonomous Robotic Systems – Lecture 4: Control Architectures", University of Nottingham, Slide 5, 2023

[2] R. A Brooks, "A Robust Layered Control System for a Mobile Robot", Massachusetts Institute of Technology, pp. 1, 1985

9. DOCUMENTATION

1. Package: minitask5

1.1 Node Name: explorer

Node "explorer" is responsible for the exploration of the map, identification and investigation of objects of interest, as well as avoiding any obstacles that the robot finds.

1.1.1 Class: odomdata

- Subscribes to /odom using Odometry.
- Class Variables:
    - prevtheta (float): the current direction of the robot from -pi to +pi (+pi = left, -pi = right)
    - prevx (float): the odometry position of the robot on the X axis
    - prevy (float): the odometry position of the robot on the Y axis
    - prevw (float): the odometry orientation of the robot on the W axis
    - quaternion (int): the angle/theta of the robot converted into quaternions.
    - firstmove (bool): a True/False statement to determine if the robot's odometry position has been initialised yet (True = uninitialised)

    - startingx (float): the starting odometry position of the robot on the X axis
    - startingy (float): the starting odometry position of the robot on the Y axis
    - distance (float): the distance the robot has travelled (difference in X and Y odometry since previous update) – to be reset frequently.
    - totaldistance (float): the distance the robot has travelled (difference in X and Y odometry since previous update) – to be reset infrequently (upon locating an object of interest, or if it is over 5) and is used to trigger "visualexploration" after the robot has travelled a long distance.
    - Increment (float): the distance the robot has travelled since its location was last updated.
- Class Methods:
    - odom_callback(self, msg): update Odometry based class variables from /odom through its subscription
    - cardinaldirection(self): determines the closest cardinal direction to the current theta (1 = North, 2 = East, 3 = South, 4 = West)

1.1.2 Class: lasers

- Subscribes to /scan using LaserScan
- Class Variables:
    - front (float): laser distance for cardinal laser at 0 degrees (North).
    - left (float): laser distance for cardinal laser at 90 degrees (West).
    - back (float): laser distance for cardinal laser at 180 degrees (South).
    - right (float): laser distance for cardinal laser at 270 degrees (East).
    - frontleft (float): laser distance for ordinal laser at 45 degrees (North-West).
    - frontright (float): laser distance for ordinal laser at 315 degrees (North-East).
    - backleft (float): laser distance for ordinal laser at 135 degrees (South-West).
    - backright (float): laser distance for ordinal laser at 225 degrees (South-East).
    - frontfrontleft (float): laser distance for 2nd intercardinal laser at 22 degrees (North-North-West).
    - leftfrontleft (float): laser distance for 2nd intercardinal laser at 67 degrees (West-North-West).
    - frontfrontright (float): laser distance for 2nd intercardinal laser at 337 degrees (North-North-East).
    - rightfrontright (float): laser distance for 2nd intercardinal laser at 292 degrees (East-North-East).
    - backbackleft (float): laser distance for 2nd intercardinal laser at 157 degrees (South-South-West).

    - leftbackleft (float): laser distance for 2nd intercardinal laser at 112 degrees (West-South-West).
    - backbackright (float): laser distance for 2nd intercardinal laser at 202 degrees (South-South-East).
    - rightbackright (float): laser distance for 2nd intercardinal laser at 247 degrees (East-South-East).
    - laserfrleft (float): laser distances for ranges from 1 to 45 degrees.
    - laserfrright (float): laser distances for ranges from 315 to 359 degrees
    - laserbkleft (float): laser distances for ranges from 135 to 179 degrees
    - laserbkright (float): laser distances for ranges from 181 to 225 degrees
    - full (float): laser distances for ranges from 1 to 359 degrees.
- Class Methods:
    - laser_callback(self, msg): updates laser based class variables from /scan through its subscription
    - checklaserinit(self): checks if lasers have been initialised – all lasers return “inf” during initialisation (“Inf” in simulation, or 0 in the real world), returns True if lasers are initialised – This should not be used in an empty world as lasers will always be “inf” as there is no detectable endpoint.
    - obstacle(self): stops the robot from moving and turning by publishing through twist, and returns True if an obstacle is detected less than 30cm away from lasers: front, laserfrright, or laserfrleft
    - backobstacle(self): returns True if an obstacle is detected less than 20cm away from lasers: back, laserbkright, or laserbkleft – Reversing is only used my move base therefore a manual readjustment is unnecessary.
    - avoidobstacle(self): whilst an obstacle is less that 30cm away from front, laserfrright, or laserfrleft, the robot will turn left through twist until there are no obstacles in front of these lasers.

1.1.3 Class: occdata

- Publishes to /map using OccupancyGrid
- Class Variables:
    - resolution (float): how many meters per pixel in the occupancy grid (set to 0.5 metres per pixel)
    - width (int): the width of the occupancy grid in pixels (set to 41)
    - height (int): the height of the occupancy grid in pixels (set to 41)
    - mymap (OccupancyGrid): stores the occupancy grid array in the OccupancyGrid data type, using resolution, width, and height.
    - rate (int): the rate that the occupancy grid should refresh.

    - xpose (int): the X odometry position of the robot within the occupancy grid
    - ypose (int): the Y odometry position of the robot within the occupancy grid
    - botrange (int): the distance within the occupancy grid from once location to another
    - grid (2D int array): a 2D array for storing and updating the occupancy grid outside of the published/subscribed dataset. (-1 = unexplored, 0 = unoccupied, 1 = occupied, 2 = not frontier eligible, 3 = green object, 4 = red object)
    - initcount (int): will be 0 if the occupancy grid has not yet been initialised to store -1's to denote an unexplored map.
- Class Methods:
    - occ_callback(self, msg): updates the mymap occupancy grid from the /map subscription
    - to_grid(self, px, py): converts an X (px) and Y (py) odometry position into an occupancy grid pixel position
    - to_world(self, px, py): converts an X (px) and Y (py) occupancy grid position into an odometry position in the real world/simulation
    - get_line(self, start, end, scandata): updates all occupancy grid positions between the start and end locations as being unoccupied, with the end position being occupied. If scandata == 3 (manual input as alternative for "Inf"), do not add an obstacle at the end position.
    - startpos(self): converts the robots current odometry position into an occupancy grid position – required for get_line and other occupancy grid functionality.
    - angle(self, theta): converts a laser angle and robot theta into an angle relative to the world rather than robot position. (0-360 degrees instead of -3.14 to 3.14)
    - endpos(self, angle, distance): uses the angle of the laser, laserscan distance, and trigonometry to calculate the odometry location of the lasers end position
    - iterateoccupy(self, las, lasangle): uses the laser angle and laser distance through get_line to update the occupancy grid
    - occupy(self): uses iterateoccupy for all specified lasers to update the occupancy grid.
    - checksurrounding(self, x, y): checks the surrounding occupancy positions of X and Y, and returns how many unknown locations it borders for usage in frontier exploration.
    - checkwalls(self, x, y): checks the surrounding occupancy position of X and Y, and returns how many occupied locations it borders for frontier exploration.

- frontierlist(self): iterates through grid positions to find any positions that border an unknown cell, and returns all locations which do.
- goaltooclose(self, x, y): checks if an occupancy grid position borders an object of interest.
- closestfrontier(self): iterates through the list returned by frontierlist, and calculates the direct distance to each frontier, and returns to closest frontier.
- mapcleanse(self): remove any placeholder values within the occupancy grid – placeholders are used to avoid investigation into certain undesirable areas.
- checkobjectsurrounding(self, x, y, itemno): check if the location borders an object of interest.
- checkobjectlogged(self, itemno): check if the endpos of a front laser scan has already been logged as an object of interest into the occupancy grid
- loggreenobject(self, objno, itemno): log the front laser endpos into the occupancy grid with the specified itemno (green) value to denote the object type. Print the object information (and location) into the command line)
- logredobject(self, objno, itemno): log the front laser endpos into the occupancy grid with the specified itemno (red) value to denote the object type. Print the object information (and location) into the command line)

1.1.4 Class: camdata

- Publishes to /camera/rgb/image_raw using Image for usage in simulation and /camera/color/image_raw for usage on the TurtleBot.
- Class Variables:
  - bridge (CvBridge): used for converting ros images to OpenCV images.
  - greenerrorx (float): centroid location on X axis for green mask image
  - greenerrory (float): centroid location on Y axis for green mask image
  - rederrorx (float): centroid location on X axis for red mask image
  - rederrory (float): centroid location on Y axis for red mask image
  - blueerrorx (float): centroid location on X axis for blue mask image
  - blueerrorx (float): centroid location on Y axis for blue mask image
- Class Methods:
  - camera_callback(self, msg): update class variables from /image_raw – uses HSV colour ranges for mask colour detection. Each mask uses a centroid for edge detection where the X and Y axis values will be updated for each colour respectively, and each mask image will be presented to the user in real time.
  - gotored(self, redobjno): if the red object has not already been logged in the occupancy grid (or unsure) move towards the red object until within range, then log the object into the occupancy grid.

- gotogreen(self, greenobjno): if the green object has not already been logged in the occupancy grid (or unsure) move towards the green object until within range, then log the object into the occupancy grid.
- avoidblue(self): if the blue centroid is within a specified vertical and horizontal range, stop moving and turn around (publish to twist) until the blue object is out of range or sight.
- bluecheck(self): check if the blue centroid (blueerrorx/y) is within a specified range, if so, return True which means that evasive action is required.
- greencheck(self): if green centroid (greenerrorx/y) is not default, and the object has not been logged, return True which means that the robot should move towards the object.
- redcheck(self): if red centroid (rederrorx/y) is not default, and the object has not been logged, return True which means that the robot should move towards the object.
- ignore(self, itemno): check if the robots grid position or neighbours are logged objects of interest, or if the cardinal direction the robot is looking has an object of interest, if so return True – ignore the object, it has already been logged and should not be investigated.

1.1.5 Class: MoveBaseActionList

- Publishes to /move_base using MoveBaseAction
- Class Variables:
  - i (int): stores how many iterations a specified process has been through.
  - iteration (int): stores 0 until all systems are operational (laser.checklaserinit) and ready for behaviours to be processed
  - freelist (list): stores the occupancy grid locations for unoccupied cells.
  - obslist (list): stores the occupancy grid locations for occupancy cells.
  - frontiertime (int): how long has the robot been attempting to navigate to a frontier.
  - movexstart (float): the X odometry position of the robot before it started a behaviour.
  - moveystart (float): the Y odometry position of the robot before it started a behaviour.
  - x (float): odometry X position of the robot
  - y (float): odometry Y position of the robot
  - robotangle (float): odometry theta orientation of the robot prior to starting a turning manoeuvre – used to check if the robot has fully completed a turn relative to its starting position.
  - dircounter (int): stores how many times the robot has moved in reverse in succession, if it is too high, corrective action will be needed to prevent the robot from unnecessarily reversing.
  - greenobjects (list): stores the location of green objects which have been found, also allows for the length of the list to identify how many green objects have been found.

    - redobjects (list): stores the location of red objects which have been found, also allows for the length of the list to identify how many red objects have been found.
    - blueencountered (int): stores how many times evasive action has been needed for blue tiles in succession – if it is too high, the robot may be stuck and will need to override its current behaviour.
    - objectfound (bool): True if an object of interest has recently been found – allows for subsequent actions to follow out of their standard order.
    - breaker (bool): True if an undesirable behaviour has occurred and that the current goal may need to be cancelled.
    - goalpost (list): the X any Y location of the next frontier, if null no frontier exists and the program should end.
- Class Methods:
    - movebase_client(self, x, y): publishes an X and Y position as a goal to movebase as part of the frontier exploration algorithm
    - tolerance(self, goalx, goaly): if the robot is within 50cm of its goal position return True – a new frontier can be generated
    - distancetogoal(x, y): calculates the direct distance from the robots position (odometry) to the goal position
    - spinaround(self, currentangle): if there is no obstacle within 20cm of the robot, turn left until it is interrupted, or a full turn has been completed. While turning, return True, if an obstacle prevents turning, return False.
    - halfturn(self, currentangle): turn left (through twist) until the odometry theta is +/- 3.1 of its starting value and then return True. Returns False while turning towards its goal. Used when the robot keeps reversing and needs to be manually turned to prevent the problem escalating.
    - currentdirection(self, previousx, previousy): depending on the odometry theta angle, determine if odometry X/Y should be increasing or decreasing while moving forwards. If the robot is suspected of reversing, return True, else return False.
    - exploration(self): while the robot is not within tolerance range of its goal, check for obstacles or objects of interest, else re-publish the goal to movebase to allow for exploration to continue for 1 second. If the robot did not move in 1 second, cancel the goal. Check for abnormal behaviours, and if applicable apply corrective behaviours.
    - objectactions(self): if a blue object is within range, take evasive action, else if a green object of interest is spotted, move towards it to log it, else if a red object of interest is spotted, move towards it to log it.
    - objectcheck(self): check if any blue, green, or red objects have been detected by the mask, return True if an object has been detected.
    - initialiseparameters(self): if initialise is 0, update startingx/y and robotangle to be odometry X, Y, and theta respectively, as well as set totaldistance to 5 to indicate that the robot should turn to look for objects.

- visualexploration(self): if the robot has travelled over 5 metres, spinaround until an object of interest has been spotted. This performs a visual check in new areas to ensure that the camera has seen all locations that the lasers have seen, as the camera does not see everything surrounding the robot.
- iterationcleanup(self): cancel the current movebase goal and update the grid goal location to be a placeholder. Cleanse the grid if needed and reset and variables which need resetting in between iterations.
- finaloutput(self): if no frontier exists, print the locations and object number for all found red/green objects. If no objects were found, state that nothing was found. Signal for shutdown as there is nowhere left that the robot can explore.