Design

My overall algorithm for my robots is fairly simple; be greedy. I chose this due to the nature of the problem and in particular the swapping mechanics. The items can be swapped between clusters and will continue to spawn into that cluster, meaning that, one could swap more value items into more favourable positions allowing for quicker collection. My algorithm is that for each robot, it will find the nearest item and collect it, then if it can find any items of higher value, collect the higher value item. If it can’t, then go home with the item it has. The change in how it is greedy is important. First looking for the nearest and then a higher value. This results in any lower value items that are in close favourable positions often getting swapped with a higher value item. This approach has three flaws that I can think of. The first flaw is that it can take a while for the higher value items to get into more favourable positions. It would probably be more efficient in the long therm put swap all the higher value items into more favourable positions before collecting any to home however I believe the my solution is has good balance of short term and long term thinking enabling it to perform well in both scenarios. There is also something to be said about having a simpler solution. The less complexity the easier it is to change the implementation without causing bugs. The second flaw is that if a robot has a lower value item but can see a higher value item that is a in a more favourable position, it will still swap it, putting the higher value item in a worse position. This is quite a major flaw and could be remedied by having a heuristic measure for how favourable the position is and comparing the positions of the two items original positions. However I did not implement this as this this is a fairly rare occurrence due to the robot is more likely to find items in the more favourable positions, Also the implementation of this fix would have difficult especially considering the inaccuracy of the item position estimation that I implemented. The third flaw is that with lots of robots this can often result in robots targetting the same or close items. I resolved this with strict management of the robots trying to make sure that they don’t target the same items or get close to each other.

If this approach was all that was needed for the robots to work. My implementation would have been a lot smaller and easier but when working with multiple robots, collision avoidance becomes a much bigger issue. This covers around half my implementation and so is important to design well. There are three main rules I designed to help my robot not collide with each other: The robots must never target a similar position, when going to the starting area they must go to one of the starting positions (where each robot spawned), when getting too close to each other, they must go to a starting position. Theoretically only the third rule is necessary as moving away to a clear location should always stop collisions. However this is very time expensive and so the other rules were introduced to make sure the location is safe and to avoid situations where the robots do get too close.

Separation of responsibility is always a difficult task and this is especially true for this problem. I have organised my solution into three components: The item filter, the robot coordinator and the robot controller,. The Item filter takes the messages from item sensor and filters out any items that the robot shouldn’t be interested in such as ones other robots are carrying and keeps ones that the robot should be interested in, the nearest of each different value. The robot coordinator takes in information about each robot and outputs information about the other robots, what places are available and if robots too close to each other where they should go to get away from each other. The robot controller, you guessed it, controls the robot. It takes in the information from the item filter and the coordinator but also the odometry and item holder information. With this information it makes a decision about where to go and tells the basic navigator form nav2 to go there. All of this could have been done just in the robot controller but this would lead to hard to maintain and understand mess. I separated it the way I did because it makes sense to have the robots to have publish and subscribe to a stand-alone node rather than to each other. This allows it to both coordinate better as it has all the information and doesn’t have a bias and take some of the workload of the robot controllers. The amount of navigation logic to put in there was a difficult decision as I wanted to it have the role of giving guidance and information rather than making the decision for the robot. This is something I didn’t totally succeed at as the coordinator does say where to move to in the case where robots are too close to each other.

Implementation

I have implemented my solution such that the robots doesn’t really set its own state but rather its state is defined by what it can see and what it is currently holding. After computing this state it then computes a target navigation goal. If that place is already being targetted it will recompute a new state and target until it finds one that isn't being targetted something that is guaranteed to happen as it can always go to a starting position.

Computing the state is pretty simple. It just follows some conditions according to this table:

note that there is a priority to the states that isn’t stated in the table in case it tries to target something that's already being targetted.

Computing the target is a little more complex. There are three different things it could be targetting: the same thing its already targetting, a item or a starting position. Out of these only the item is a little complex. It uses the item’s diameter to estimate a distance and the item x value to estimate an angle which it can then use to compute the coordinate relative to the robot. This was made complex due to the camera not being directly centre of the robot meaning that certain bearing would look different depending on the angle you look at it with but more importantly the that the item’s x value compared to the angle of the item will change depending on distance.

The coordination between robots was probably the hardest to implement. I started with no robot targetting the same same position. This is done by all robots publishing what position they are targetting and the coordinator collecting them all and republishing them all with the respective robot id. The each robot can then individually figure out if a robot is already targetting the area they were about to target and then choose a different target. For this problem this is very inefficient. It would better if the robots collected each others targets directly, as there are less messages and also wouldn't need to wait for the coordinator. I decided against doing this as I knew I would need the targetting information for the other rules and the time difference isn’t large enough to make a great difference. For the starting position rule the robots all publish their starting pose. The coordinator subscribes to this and develops a list of all starting positions. It then filters these based on if any robot is targetting a starting position to publish the list of available starting position that a robot can choose from. This gives the robots more options rather than requiring them to go back to their own starting position. This is more suited to the coordinator then having it in the robot controller as its more of a centralised list that all robots must follow even if they could work it out by themselves. The final rules about when they get too close to each other works by each robot sending updating the coordinator with their positions. The coordinator subscribes to this and maintains an list of their positions. It then goes through the list and if any are too near to each other it computes the near available starting position that is nearest to the opposite bearing to the average bearing of the near robots and publishes the position the robot should go to. This is the most suited to the coordinator as if each robot did it individually they could end up targetting the same starting position before the available list of starting positions is updated.

I didn’t use any actions in my implementation, instead only using topics to communicate across nodes, as I don’t believe any of the communication really fit the use of actions. The available starting positions had potential as robots could have used requested the most up-to-date list but that doesn’t really fit the use case for actions. The most suitable would probably be the going to a starting position after getting too near another robot. As it could be used to make sure they do move away from each other. If I had decided to have all the decision and targetting logic in the coordinator instead of the controller there definitely would be a use case for actions to keep track of the robots as they move towards their targets.

The robots needed to know the their id so so that they can send it along with the messages they send, so that the know which public messages are addressed to them and so they can tell which item they are holding. I gave them this information through the use of parameters so that each automatically knows. This also allows the robot controller to use the came callback for each robot as it can distinguish between robots based on their id and then add it back when sending a message to that robot.

Analysis

Evaluation

A strength of my solution is that the speed at which ‘value’ is collected increases with time while having a decent collection rate at the start as well. It out performs the turtlebot3\_gazebo.turtlebot3\_drive solution, which granted isn’t much of a feat, by a lot.

The design of my solution is simple and easy to understand making it easier for other developers to improve upon.

The design accounts for multiple robots well and offers robust and efficient robot avoidance strategies.

The estimation of the items works very well, the

I believe my design is very strong but my implementation has a few flaws.

(shown by the rviz visualisation not moving perpendicular to the robot).

In a real world scenario my solution would perform terribly. The main reason is that it uses the odometry to get its location and bearing. This only works in simulation as there is no wheel slipping which would result in odometry drift. In reality the position estimate would slowly become worse and worse over time until the robot thinks its in a place that its not. This could be improved by using a different localisation technique such as using the location nav2 thinks the robot is.

In reality it is also not guaranteed that you have a map of the area. This would impact the navigation of the robot somewhat, although it nav2 should be able to deal with that problem.

Another potential problem is the gradient of the ground. In simulation the robots have only been tested on a flat surface. The filtering out of items that a different robot is carrying might work against us if the gradient changes as items may appear above the robot. The robots also might behave poorly on gradient e.g. failing to stop in time. You might be able account for a static gradient if you can detect it but one that changes could mess with the positions of the items too much.

The simulation camera acts differently to real world lens. It has no lens distortion which is why I could estimate the angle of the items in regards to the robot so accurately. This would break if using a real world lens and having a decent location estimate for the items is the basis of my solution. With enough maths this can be accounted for. An image can be effectively flattened if the distortion of the lens is known.

Safety and Ethics