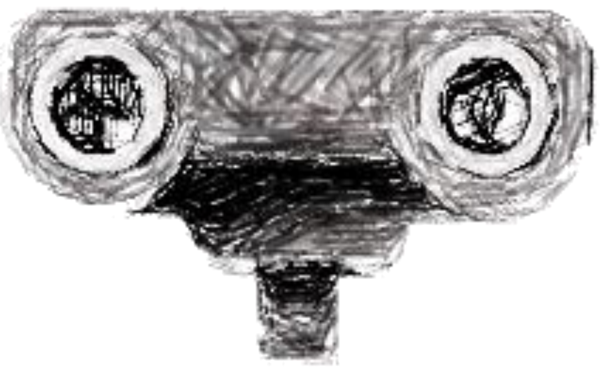
G54ARS Report

Barnabas Forgo
& Faisal Syed

Group A7



Introduction

In this assignment our task was to implement four behaviours using Brooks' subsumption architecture [1] on a Lego EV3 robot. This architecture uses a stack of behaviours that run in parallel. Each behaviour produces an output of actuator commands and an active/inactive signal. An additional arbiter task is used to choose which commands should be sent to the actuators. The arbiter always chooses the actuator commands from the highest ranking active behaviour in the stack. This way higher level behaviours can '*subsume*' lower level behaviours' actuator commands. The four behaviours that were implemented are the following (from lowest to highest level): *forage*, *follow*, *avoid*, and *observe*.

1. The first behaviour in the stack is *forage*. This task is the simplest; its only job is to wander around to find the line in order to activate the *follow* behaviour. We have chosen to implement this as a fixed pattern of moving in a straight line.
2. The next behaviour in the stack is *follow*. This behaviour uses the colour sensor on the top of the robot to detect and follow a black projected line. We implemented this using a simple bang-bang controller that swing turns either left or right, with a dynamically adjusted setpoint.
3. The third behaviour for our robot is *avoid*. This task takes input from the two ultrasonic sensors in the front of the robot to avoid any obstacles that might be on the line. Our robot does this by following a trapezoidal path around the obstacle.
4. The highest-level behaviour implemented was *observe*. This task should detect the longest straight of the projected track and halt the robot in the middle of it. We implemented this, by using an estimation of the location of the robot, to build a map of the track. Once this map is built, the data is analysed, in order to find the target location during the next lap around the tack.

Discussion of Behaviours

Forage

This behaviour is the simplest of the four, only making the robot move in a straight line. *Forage* is always active, as it cannot subsume other behaviours, and serves as a fallback for the others. There are a few reasons for choosing such a simplistic behaviour:

- ☐ Firstly, it is deterministic, which makes debugging easier.
- ☐ Secondly, if the robot is started inside the track then it is guaranteed to find the line.
- ☐ The third reason is that the edge of the projection and the line is indistinguishable for the robot. Therefore, if e.g. a random walk was implemented then it would be likely to hit the edge of the projection, which would be unrecoverable.
- ☐ Lastly, this was the first behaviour implemented, and therefore simplicity was key. Towards the final lab sessions, we considered implementing a spiralling motion, but we opted to fine-tune the other behaviours instead due to a lack of time.

Clearly, this type of motion has downsides, the main one being is that it is impossible for the robot to find the line if it is not facing the track, when the program is started. Furthermore, if the robot does find the line but at a steep angle, then it will be a lot harder to start *follow*.

Follow

For us, the hardest task was to implement a reliable line following algorithm. We tried many different methods of achieving this, ranging from a simple fixed threshold bang-bang controller, to a PID controller, and even a Fuzzy Inference System. Finally, we achieved a fairly reliable solution using a bang-bang controller combined with dynamic threshold calibration.

What's common in all of these implementations is that we are not truly following the line, but the edge of it. This is a due to the limitations of the robot. If the robot was to follow the centre of the line, then it would be impossible to know (with only one colour sensor) which way can it return to the centre of the line when it wanders off. This would mean that the robot would only be able to follow the line in one direction, and often it would turn around and repeat some parts of the track. On the other hand, if the robot follows the 'grey area' between the black line and the white background, then it is easy to tell which way to correct in turns: when it detects more dark colours turn right, more bright colours then left.

Our first major attempt of achieving this was using a PID controller. We tried several ways to use the PID controller: fixed setpoint, dynamic setpoint, different error calculations, various tuning methods, etc. However, it was always either oscillating too much, or not responding to tight turns enough.

As an alternative we implemented a type-1 Fuzzy Inference System based on Juzzy [2]. Our system had two inputs, the current light reading and the previous one; and one output, differential motor command as a single value (i.e. negative for left turn, positive for right turn), resembling a PD controller. The first problem we encountered with this is the computational limitation of the robot. We increased the task priority for the *follow* behaviour which fixed this problem, but we quickly gave up on this method as tuning a FIS controller proved even harder than a PID, and we did not have the time to perfect it.

Finally, we tried a simpler bang-bang controller, but with reusing the dynamic setpoint logic from our previous PID controller. This controller can be modelled as a Finite State Machine, as seen in Figure 1 on the right. In this algorithm we use sliding windows to automatically calibrate the colour sensor. A main window of size 10 is used to collect sensor samples continuously. Every time the window has "rolled over" the lowest and highest values are used to populate two other sliding windows. This way both the darkest and brightest light readings are stored in memory and the robot can adjust its setpoint.

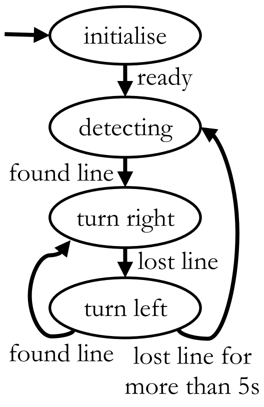


Figure 1 Follow behaviour represented as a Finite State Machine

One problem we faced was that, if the robot is following the line too closely, then it would "forget" about the bright values and miscalibrate itself. To tackle this, we added rules for the maximal low light values, and the minimal high light values. Furthermore, the high and low windows would only be updated if the new candidate values are different enough, i.e. when the robot is reacting to changes in lighting conditions. However, what ultimately solved this problem was using a more "shaky" but reliable controller. This way the robot is continuously exposed to bright and dark lights and it can accurately follow the line, albeit with considerable oscillation.

	A	B
old	0%	5%
	0%	5%
	0%	5%
	0%	5%
	0%	5%
	0%	5%
	0%	10%
	5%	10%
	15%	10%
new	80%	40%

Table 1 Induced Ordered Weighted Average weights used for sensor sampling and calibration

In order to utilise the collected values, we used Induced Ordered Weighted Average where ordering was determined by the recency of a reading. We used this method as it requires no transformation of the sliding window, and when fusing sensor data over time the most recent values are the most important. We used two sets of weights for sampling the sliding windows, which can be seen in Table 1 on the right. Weights *A* were used to determine the current light sensor reading. These weights are heavily biased towards new values, disregarding any values that are older than three readings. Weights *B* were used to sample the low and high windows. These weights use all of the values in the windows, however it is still biased towards new readings, in order to react to sudden changes. To form a setpoint, we used a weighted average over the high and low values, and through trial and error we chose the weights to be 50%-50%.

Clearly, this algorithm is not perfect, and has many shortcomings. Firstly, it oscillates a lot, which is undesirable, but this was the only way that setpoint calibration could be implemented reliably. Secondly, the controller uses fixed, low speeds to turn the robot, which means that it cannot take very sharp turns. We could have fixed this, by retuning the controller weights for a higher turning speed. Lastly, the corners of the projection still confuse the robot sometimes, as there, the bright and dark values have the lowest difference. Despite these negatives, this behaviour performs fairly well, as it could follow the line even when the Sun was interfering with the projection.

Avoid

To implement the *avoid* behaviour we used the two ultrasonic sensors in the front of the robot. The logic for this behaviour can be represented as a Finite State Machine which can be seen below, in Figure 2. This algorithm can be explained as follows:

- ☐ Detect object. Behaviour is inactive at this point so that line following can continue.
- ☐ When object is detected, turn towards the further reading of the two sensors.
- ☐ When object is no longer in sight move forward two robot lengths, to turn safely.
- ☐ Turn in the opposite direction of the first turn.
- ☐ When object is in sight again turn back slightly so that the robot can move forward.
- ☐ Move forward two robot lengths.
- ☐ Turn until object is detected, then turn back slightly.
- ☐ Move forward one robot length, or until the line is detected again.
- ☐ Repeat.

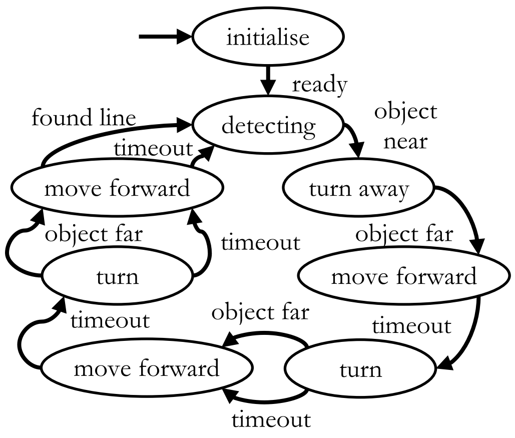


Figure 2 Avoid behaviour represented as a Finite State Machine

In effect, this algorithm moves the robot around the obstacle on a trapezoidal trajectory, in the optimal case. However, due to its simplicity it can fail at many points. The main issue is when the robot cannot detect the object on the return path, due to sensor noise. In this case it usually just ends up hitting the obstacle as it ignores sonar readings in the straight parts of the trapezoid. Furthermore, it is hard for it to return to line following, as more often than not, it arrives at the opposite end of obstacles at sharp angles.

We realise that this is not the most robust behaviour, however we focused more on the *follow* and *observe* behaviours. One way this algorithm could be improved is to use a sliding window to sample the sensors, similarly to line following. As it stands, this behaviour takes every sonar reading as ground truth, even though sonars are very liable to sensor noise. Another improvement could be to use the coordinate system from the observe behaviour to more accurately pinpoint the location and size of the obstacle, in order to plan the avoid path.

Observe

Our implementation of the observe behaviour has three consecutive stages, as can be seen in Figure 3. The first stage is *exploring*. In this state the robot goes around the lap once, to collect information about the track. To achieve this, we used a dead reckoning algorithm, where we maintain the pose (x, y, θ) of the robot in a relative coordinate system using the motor encoders and the gyroscope (weighted towards the gyro). As the robot moves along the track, samples are taken of the current pose. This happens every time the angle of the robot changes by more than 15° since the last sample. Over time this data builds a graph of the race track. Once this graph is "closed", i.e. when the robot has returned to its starting position after one lap, the robot advances to the next stage.

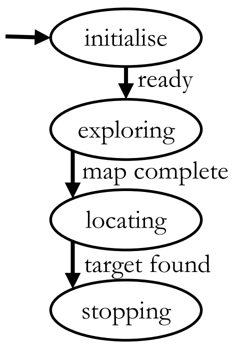


Figure 3 Observe behaviour represented as a Finite State Machine

In the second, *locating* step, the generated graph is analysed to find the longest edge. The target is set to the midpoint of this section. The robot continues to follow the line until it gets close enough to this target. After this, the robot moves to the last stage of the behaviour, that halts the robot, subsuming other motor commands.

There are three key issues with this solution, that we did not have time to fix. The first one is that it does not implement the requirements of the behaviour, which is observing the follow task. This means that the graph map is updated even when the robot is not following the line. The second issue is that there is no post-processing applied to the graph. Therefore, if it splits a straight section of the longest track into multiple edges, then it will not find the midpoint of it as it only looks at graph edges. This could be solved by applying an edge contraction algorithm to the graph map. Lastly the location estimation of the robot is inaccurate, so using two laps to form the map could have resulted in more robust behaviour.

Interaction of Behaviours

The beauty of the subsumption architecture is that it combines simple, well defined behaviours to form complex behaviour, and this assignment demonstrates this perfectly. How this architecture works is discussed in the Introduction, and our implementation follows this very closely. The standard representation of the architecture can be seen on the right in Figure 4. Even though these behaviours can mostly function independently, there are some interesting interactions when they run in parallel.

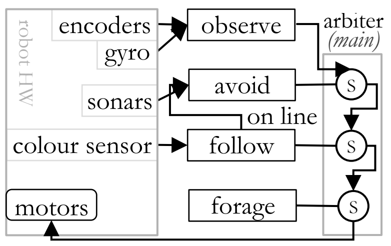


Figure 4 Subsumption architecture model diagram

- ☐ Since the *avoid* task always takes more than 5s therefore, it will always force the *follow* behaviour to "give up" and return to *forage*. This means that the *avoid* behaviour will always be followed by *forage* if the line was not passed, or by *follow* if it was.
- ☐ Another interaction between *avoid* and *follow*, is that *avoid* usually spends a considerable amount of time without touching the line. This causes the sliding windows in *follow* to "forget" about the low values. Even though this is corrected soon after hitting the line, it makes it hard to return to *follow*.
- ☐ The *observe* behaviour can subsume all of the below layers. This makes it possible to halt the robot completely, by sending 0-0 motor commands. If a behaviour fusion architecture was used, then implementing this would be considerably harder.
- ☐ Due to the limitation of *observe*, if *avoid* interferes with map making then the robot might never halt.

Reflection

Looking back at the assignment, I was rather disappointed, because most, if not all coding was done by myself. This was due to the fact, that I have high code quality standards, that I can not expect others to follow. However, I have tried to involve my partner, by asking him for ideas, or research, but he refused to cooperate on several occasions. When we were working on the assignment in the lab, I explained the code I was writing, however he did not seem to be able to follow advanced concepts, that were covered in the lectures. I think that my learning experience could have been a lot better if my peer was contributing with ideas and critiquing mine. Overall, I tried my best to work better together, but my efforts were hindered by a lack of willingness to cooperate from my partner. Looking back, I think I might have made my team mate feel uncomfortable with contributing, as I have dismissed some of his earlier ideas that did not meet my expectations, and this is something I would like to improve on in future projects.

Conclusion

Overall, we have developed a reliable subsumption architecture, that is easy to extend, and follows the original design very closely. We have also developed a few more generic data structures and algorithms that can be reused in different projects; e.g. PID controller with on-the-fly calibration, sliding windows with ordered weighting, utility functions, and a rudimentary Fuzzy Inference System. Even though we tried to fine-tune the individual behaviours, there is still plenty of work to be done:

1. The *forage* behaviour works fine if its precondition is met: facing the track. An improvement to this task could be to implement a spiralling, or hill climb/descent type of motion.
2. Our *follow* behaviour is fairly robust, even in changing lighting conditions, due to its automatic sensor calibration logic. However, a key limitation is that it requires a constant exposure to bright and dark colours to achieve this. This is why the robot needs to oscillate, and also this is why it is hard to return to *follow* after *avoid*. Potentially, these issues can be resolved by further tuning how the calibration windows are updated and queried.
3. The *avoid* task is probably the one that needs the most work. This behaviour could be improved by performing path planning instead of a semi-fixed path. To achieve this, the location of the robot from the *observe* task could be used to enhance the map with obstacle information.
4. Finally, the *observe* behaviour has produced good results, even though it does not match the specifications completely. To make it compliant the map update process needs to be conditional on the output of *follow*, i.e. location samples can only be taken while on the line, w.r.t. the dynamic threshold. Another improvement could be to separate the localisation and mapping parts. If the localisation part was a lower level behaviour (without motor control), then other behaviours could use the position of the robot in their decision making, or make maps of their own.

Bibliography

[1] R. Brooks, 'A robust layered control system for a mobile robot', *IEEE journal on robotics and automation*, vol. 2, no. 1, pp. 14–23, 1986.

[2] C. Wagner, 'Juzzy-a java based toolkit for type-2 fuzzy logic', in *Advances in Type-2 Fuzzy Logic Systems (T2FUZZ), 2013 IEEE Symposium on*, 2013, pp. 45–52.