Intelligent and Mobile Robotics - 2nd project

Development of an agent for a robot that solves labyrinths

Vedran Semenski

Abstract **- This paper presents and describes an implementation of an reactive robotic agent using a library of developed behaviours, all structured in a subsumption architecture. Unlike the 1st project which was tested on a simulator this one was tested on a real physical robot. This agent was developed for a project that is part of the "Intelligent and mobile robotics" course and is one of three in total.**

I. INTRODUCTION

The goal of this project is to understand the development mechanisms of microcontroller-based robots, understand the differences between the development of simulated robotic agents and robotic agents based on microcontrollers, use the sensors of a real robot and understand its functioning and limitations; develop techniques and methods to mitigate the impact of these limitations on the overall behaviour of the robot [1], further explore the notion of a reactive robotic agent, improve upon and further develop the behaviours from the 1st project and test the solution an a real physical robot provided by the university. This solution was developed using C in basic text editor in a Linux Ubuntu operating system. All aspects of the development, behaviours and test results will be presented and explained in this document.

II. OBJECTIVES

In this project 2 main objectives were determined:

1. Develop at minimum this of basic behaviours for navigation in a labyrinth and demonstrate them separately: "Obstacle avoidance", "Wall following", "Move towards the beacon" and "Stop at the beacon area". [1]
2. Develop a robotic agent able to meet both goals of the Micro-Mouse competition, that is, locate the beacon area, pass over it, signalling the fulfilment of the first goal, and return back to the starting position, stopping as close as possible to that position. The agent has to be demonstrated using two different labyrinths, one whose layout is known in advance and another that will only be revealed after the executable code has been uploaded to the robot. [1]

III. ROBOTIC AGENT

In this case particular case the robotic agent is is an autonomous piece of software that interacts with the outside world by reading sensor readings and controlling motors that enable the robot to move.

The solution communicates with the robot via a library provided by the project supervisors. After compiling the solution with the provided compiler ("pcompile") the generated hex file is uploaded to the robot using a USB cable and the robot is ready for testing.

Usage of the libraries functions is abstracted and separated into 3 basic parts. In the main function the initialisation calls are made. In the *sensors.c* file all the communication regarding sensors and sensor data is handled and all sensor data is stored in custom structures. Basic movements regarding the left and right motors are handled in the *movoments.c* file. All of the behaviours can be seen in the *behaviors.c* file. This has been done so the solution could be modular meaning that changing a library for communicating with the robot (like sensor functions of movement functions) doesn't necessarily mean making a lot of changes in the agent.

IV. SUBSUMPTION ARCHITECTURE

The subsumption architecture implies two main things. The first one is that all of the agents behaviours should be organized in such a way that every behaviour has a certain set of conditions that have to be satisfied if the behaviour is to be executed. The second one is that all of the behaviours should be organized in a prioritized list manner and the behaviour that is executed must be the one with the highest priority from a set of behaviours which have satisfied their set of behaviours. [2]

This architecture is implemented by organizing behaviours in separate functions inside a seperate *"behaviors.c"* script. An array of behavior IDs is organized in a prioritized manner. Each behavior is first tested with a "test_behavior_name" function and the behaviour with the highest priority is than executed. It should also be mentioned because of easier implementation a priority value of 0 is the highest priority and priority value of 5 is lower in priority than 4.3.2 etc. so a bit contra intuitive at first. This way of executing behaviours is completely in compliance with the subsumption architecture. The solution is written in such a way so it is easily reconfigurable and enables the

programmer to easily modify the way behaviours are tested and executed. One benefit of this layout is exploited in a way that the behaviours aren't always prioritized in the same way and not all behaviours are available for testing and execution at all times. This will be explained further in this document.

V. BEHAVIOURS

One thing that should be mentioned before the robots moving behaviours is the way the beacon sensor is configured to work. The beacon sensor saturates if it is locked on to the beacon for a long period of time (1-2 seconds). To resolve this problem the beacon sensor servo is ran in such a way that it goes from one extreme position (-180◦) to the other (+180◦) until it detects the beacon. This angle is stored as the beacon direction and then the servo moves from left to right of the beacon (about +/- ~30◦), correcting the beacon direction every time it detects the beacon and continues to track it until it loses it.

Every behaviour is associated with its test, reset and execution function. All of them can be found in the *behavior.c* file. The behaviours are organized in a prioritized list but that list isn't static. The priority of the behaviours changes depending on the situation. This i important because this is a major improvement comparing to the solution from the 1st project.

List of behaviours:

- AvoidColision - "Obstacle avoidance"
- StopAtBeacon - "Stop at the beacon area"
- FollowBeacon - "Move towards the beacon"
- FollowWall - "Wall following"
- Wounder
- StopAtStartingPosition
- FollowStartingPosition

AvoidColision behaviour activates when the robot comes very close to an obstacle. The actions consists of the robot stopping and rotating itself around its axis in the direction of the side that is more free of obstacles. If the right obstacle distance sensor shows that there is an obstacle closer to that side compared to the left sensor the robot will turn left and vice versa.

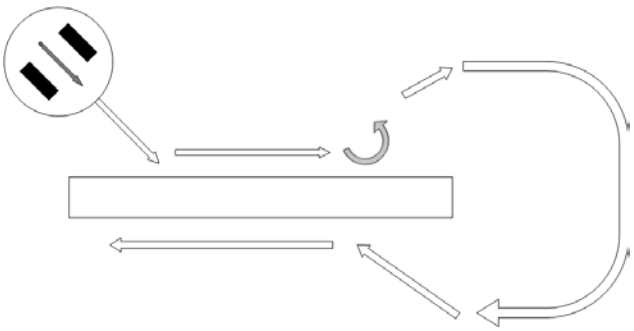
StopAtBeacon activates when the ground sensor shows that the robot arrived to the beacon area. The behaviour just consists of the robot stopping.

StopAtStartingPosition activates when position of the robot is approximately equal to its starting position. The behavior just consists of the robot stopping.

FollowBeacon activates when the beacon is visible by the robot. The behaviour consists of moving in the direction of the beacon.

FollowStartingPosition can be activate basically at any time after the robot moves away from its starting position. The behaviour consists of returning to the starting position. It achieves this by following a path of "breadcrumbs" that it left behind while searching for the robot. While the robot is searching for the beacon it remembers its position every 20-50 cycles (depending on the settings) of reading sensor data and it uses that position history data to return to the starting position.

FollowWall behaviour activates when the sensors indicate that there is an obstacle near the robot. The behaviour consists of the robot maintaining the distance between itself and the obstacle and moving alongside it. When the robot senses that the edge of the robot is near it rotates away from the wall about 30 degrees and than starts moving in a circular curve until it sees the wall again on the other side of the corner. This behaviour remembers the side on which the wall is located so it if not reset or interrupted this behaviour would make the robot go around the same wall forever. This behaviour can make the robot go around sharp turns and shallow turns. If the robot detects an inside turn it just stops and rotates in the direction of the turn until it can continue to go forward.



Picture 1. FollowWall behaviour

Wander behaviour has no conditions that need to be satisfied for the behaviour to activate. This behaviour has the lowest priority of all so it always has to be available for execution. For testing purposes this behaviour is implemented so it just makes the robot move in a straight line.

This set of behaviours would have a lot of problems working together in the goal of finding the beacon area and returning to the starting point if not managed in some way. The priority list has many possibilities but these are the main ones used in this solution

Initial (default) priority list:

1. AvoidColision
2. StopAtBeacon
3. FollowBeacon
4. FollowWall
5. Wounder

This list of behaviours has a simple problem with the *FollowBeacon* and *FollowWall* behaviors. A problem that

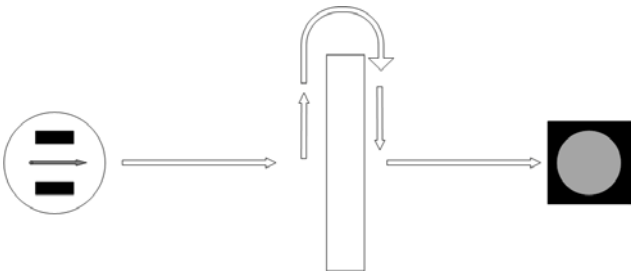
occurs is that the FollowWall behavior gets interrupted by the *FollowBeacon* behavior and that one doesn't work if there is a wide "U" shaped obstacle in the way between the robot and the beacon so it is interrupted by the *AvoidColision* until it loses sight of the beacon and the *FollowWall* behaviour is engaged. There are many potential loops and complications that can derive from this kind of behaviour. To solve this problem the list changes when it detects that the *FollowBeacon* can be executed and there is a wall between the beacon and the robot. After detection the *FollowWall* behavior is promoted over the *FollowWall* behavior.

Priority list after promotion:

1. AvoidColision
2. StopAtBeacon
3. FollowWall
4. FollowBeacon
5. Wounder

This continues until the robot reaches a point that is on the line between the beacon and the point when the behaviour was promoted. After that the *FollowWall* behaviour list is returned back to the initial state.

This results in a very good overall behaviour of the robot. The resulting global behaviour should do the following. It should follow the beacon if it sees it and if it detects an obstacle between it and the beacon it should follow it until it reaches the same point on the other side of the obstacle and continue its journey to the beacon.



Picture 2. resulting overall behaviour

Another situation where the priority list changes drastically is when the beacon area is reached. After that has happened the *StopAtBeacon* and *FolowBeacon* behaviours are not useful anymore. Because of this these behaviours are exchanged with the *StopAtStartingPosition* and *FollowStartingPosition* behaviours.

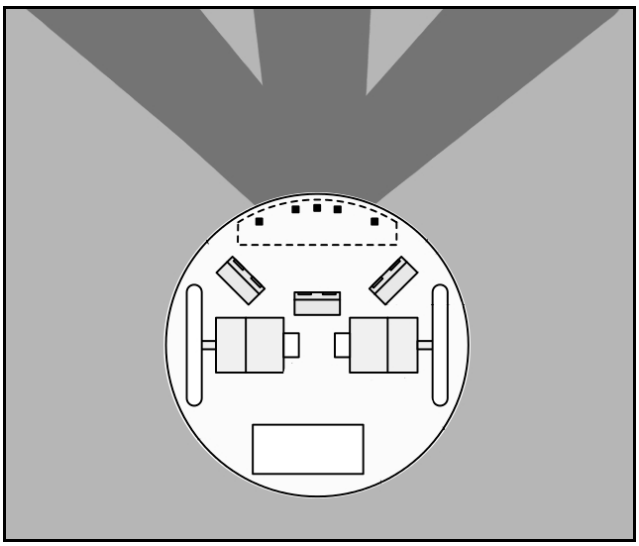
Priority list for returning to the starting point:

1. AvoidColision
2. StopAtStartingPosition
3. FollowStartingPosition
4. FollowWall
5. Wounder

The same problem with the *FollowWall* and *FollowStartingPosition* behaviours occur as previously with the *FollowWall* and *FolowBeacon* behaviours so the same solution is implemented by promoting the *FollowWall* behaviour.

VI. TESTING

Test results indicate that this agent has a very small tendency of colliding with walls and getting stuck in a loop. It found its way to the beacon area on all tested maps almost always and returned back to its starting position as well. Testing was done on the provided robot at the laboratory on various labyrinth setups. One of the aspects that we are supposed to present is also demonstrating 4 key behaviours independently from one another. This aspect was tested and the results were very promising and showed that the behaviours were very reliable. A problem can occurs only with the *FollowWall* behaviour because the robot has a limited field of view so it goes around corners "blindly" (the IR sensors for detecting distance from obstacles do not have a wide enough field of view) and depending on the conditions and the level of noise the robot can scrape a corners edge.



Picture 3. an approximation of the robots field of view

VII. Conclusion

The subsumption architecture provides a very flexible platform for expanding the behaviour library and further developing and improving the agent. It also enables easy tuning, especially with a large library of behaviours. This was exploited in with the dynamic priority list explained earlier. The reactive aspect of the agent simplifies the implementation but proves to result in a very satisfying behaviour of the agent. The second benefit of the reactive aspect is that the agent does not waste time on complex calculations and as a result minimizes the time from receiving data from the sensors and the time of executing the behaviours.

By the very nature of this kind of setup the agent has no learning aspect and does not remember any aspect of the environment it is placed in and does not adapt accordingly. As a result this setup is limited in changing and improving the agent in a major way.

Having that in mind it is still a very simple and fast way to get good results and should be used in conjunction with other solutions for an even better result.

This project has been very insightful in the area of robotics, microcontrollers, and dealing real world conditions with sensor readings (noise and inconsistency). It has also provided me with a new experience from which I most definitely benefited as a software engineer.

References

[1] -"Robótica Móvel e Inteligente, 2014/15, Project 2, Development of an agent for a robot that solves labyrinths", http://elearning.ua.pt/pluginfile.php/224423/mod_resource/content/2/RMI_1415_project2.pdf, October 2014.

[2] -" Inteligent and Mobile Robotics - 2nd project Development of an agent for a robot that solves labyrinths", Vedran Semenski.